

(12) NACH DEM VERTRAG ÜBER DIE INTERNATIONALE ZUSAMMENARBEIT AUF DEM GEBIET DES  
PATENTWESENS (PCT) VERÖFFENTLICHTE INTERNATIONALE ANMELDUNG

(19) Weltorganisation für geistiges Eigentum  
Internationales Büro



(43) Internationales Veröffentlichungsdatum  
27. Februar 2003 (27.02.2003)

PCT

(10) Internationale Veröffentlichungsnummer  
**WO 03/017095 A2**

(51) Internationale Patentklassifikation<sup>7</sup>: **G06F 9/445**

(21) Internationales Aktenzeichen: PCT/EP02/10065

(22) Internationales Anmeldedatum:  
16. August 2002 (16.08.2002)

(25) Einreichungssprache: Deutsch

(26) Veröffentlichungssprache: Deutsch

(30) Angaben zur Priorität:  
101 39 170.6 16. August 2001 (16.08.2001) DE  
101 42 903.7 3. September 2001 (03.09.2001) DE  
101 44 732.9 11. September 2001 (11.09.2001) DE  
101 45 792.8 17. September 2001 (17.09.2001) DE  
09/967,847 28. September 2001 (28.09.2001) US  
101 54 260.7 5. November 2001 (05.11.2001) DE  
102 07 225.6 21. Februar 2002 (21.02.2002) DE  
PCT/EP02/02398 5. März 2002 (05.03.2002) EP  
PCT/EP02/09131 15. August 2002 (15.08.2002) EP

(71) Anmelder (für alle Bestimmungsstaaten mit Ausnahme von US): **PACT XPP TECHNOLOGIES AG** [DE/DE]; Muthmannstrasse 1, 80939 München (DE).

(72) Erfinder; und

(75) Erfinder/Anmelder (nur für US): **VORBACH, Martin** [DE/DE]; Gotthardstrasse 117a, 80689 München (DE). **MAY, Frank** [DE/DE]; An der Tuchbleiche 12, 81927 München (DE). **NÜCKEL, Armin** [DE/DE]; Drosselweg 4, 76777 Neupotz (DE).

(74) Anwalt: **PIETRUK, Claus, Peter**; Heinrich-Lilienfein-Weg 5, 76229 Karlsruhe (DE).

(81) Bestimmungsstaaten (national): AE, AG, AL, AM, AT (Gebrauchsmuster), AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ (Gebrauchsmuster), CZ, DE (Gebrauchsmuster), DE, DK (Gebrauchsmuster),

DK, DM, DZ, EC, EE (Gebrauchsmuster), EE, ES, FI (Gebrauchsmuster), FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, OM, PH, PL, PT, RO, RU, SD, SE, SG, SI, SK (Gebrauchsmuster), SK, SL, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, ZA, ZM, ZW.

(84) Bestimmungsstaaten (regional): ARIPO-Patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), eurasisches Patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), europäisches Patent (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, SK, TR), OAPI-Patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

**Erklärung gemäß Regel 4.17:**

— hinsichtlich der Berechtigung des Anmelders, ein Patent zu beantragen und zu erhalten (Regel 4.17 Ziffer ii) für die folgenden Bestimmungsstaaten AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, OM, PH, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TN, TR, TT, TZ, UA, UG, UZ, VC, VN, YU, ZA, ZM, ZW, ARIPO-Patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), eurasisches Patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), europäisches Patent (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, SK, TR), OAPI-Patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG)

**Veröffentlicht:**

— ohne internationalen Recherchenbericht und erneut zu veröffentlichen nach Erhalt des Berichts

Zur Erklärung der Zweibuchstaben-Codes und der anderen Abkürzungen wird auf die Erklärungen ("Guidance Notes on Codes and Abbreviations") am Anfang jeder regulären Ausgabe der PCT-Gazette verwiesen.

(54) Title: METHOD FOR THE TRANSLATION OF PROGRAMS FOR RECONFIGURABLE ARCHITECTURES

(54) Bezeichnung: VERFAHREN ZUM ÜBERSETZEN VON PROGRAMMEN FÜR REKONFIGURIERBARE ARCHITEKTUREN

(57) Abstract: The invention relates to data processing with multidimensional fields and high-level language codes which can be used advantageously therefor .

(57) Zusammenfassung: Die Erfindung betrifft die Datenverarbeitung mit multidimensionalen Feldern und gibt an, wie hierfür vorteilhaft Hochsprachencodes einsetzbar sind.

WO 03/017095 A2

**Titel:** Verfahren zum Übersetzen von Programmen für rekonfigurierbare Architekturen

### **1. Einleitung**

Die vorliegende Erfindung betrifft das Oberbegrifflich Beanspruchte. Damit befaßt sich die vorliegende Erfindung mit der Frage, wie rekonfigurierbare Architekturen optimal verwendet werden können und insbesondere damit, wie Anweisungen in einer gegebenen Hochsprache in rekonfigurierbaren Architekturen optimal zur Ausführung gebracht werden können.

Um in sog. Hochsprachen geschriebene Anweisungen zur Handhabung von Daten (Programme) in einer jeweiligen, zur Datenhandhabung verwendeten Architektur zur Ausführung zu bringen, sind sog. Compiler bekannt, die die Anweisungen der Hochsprache in an die verwendete Architektur besser angepaßte Anweisungen übersetzen. Compiler, die dabei hochparallele Architekturen besonders unterstützen, sind demgemäß parallelisierende Compiler.

Parallelisierende Compiler nach dem Stand der Technik verwenden für gewöhnlich spezielle Konstrukte wie Semaphore und/oder andere Verfahren zur Synchronisation. Dabei werden typischerweise technologiespezifische Verfahren verwendet. Bekannte Verfahren sind nicht geeignet, um funktional spezifizierte Architekturen mit dem zugehörigen Zeitverhalten und imperativ spezifizierte Algorithmen zu kombinieren. Daher liefern die verwendeten Methoden nur in Spezialfällen zufriedenstellende Lösungen.

Compiler für rekonfigurierbare Architekturen, insbesondere für rekonfigurierbaren Prozessoren, verwenden für gewöhnlich Makros, die speziell für die bestimmte rekonfigurierbare Hardware erstellt wurden, wobei für die Erstellung der Makros zumeist Hardwarebeschreibungssprachen wie z.B. Verilog, VHDL oder System-C verwendet werden. Diese Makros werden dann von einer gewöhnlichen Hochsprache (z.B. C, C++) aus dem Programmfluss heraus aufgerufen (instantiiert).

Compiler für Parallelrechner sind bekannt, die auf einer grobgranularen Struktur, zumeist basierend auf kompletten Funktionen oder Threads Programmteile auf mehrere Prozessoren abbilden.

Weiterhin sind vektorisierende Compiler bekannt, die eine weitgehende lineare Datenverarbeitung, wie z.B. Berechnungen großer Ausdrücke in eine vektorisierte Form umwandeln und damit die Berechnung auf superskalaren Prozessoren und Vektorprozessoren (z.B. Pentium, Cray) ermöglichen.

Vorliegend wird ein Verfahren zur automatischen Abbildung von funktional oder imperativ formulierten Rechenvorschriften auf unterschiedliche Zieltechnologien beschrieben, insbesondere auf ASICs, rekonfigurierbare Bausteine (FPGAs, DPGAs, VPUs, ChessArray, KressArray, Chameleon, etc.; im folgenden unter dem Begriff VPU zusammengefaßt), sequentielle Prozessoren (CISC-/RISC-CPUs, DSPs, etc.; im folgenden unter dem Begriff CPU zusammengefaßt) und parallele Rechnersysteme (SMP, MMP, etc.). Hingewiesen wird insbesondere in diesem Zusammenhang auf die folgenden Schutzrechte und Patentanmeldungen desselben Anmelders: P 44 16 881.0-53, DE 197 81 412.3, DE 197 81 483.2, DE 196 54 846.2-53, DE 196 54 593.5-53, DE 197 04 044.6-53, DE 198 80 129.7, DE 198 61 088.2-53, DE 199 80 312.9, PCT/DE 00/01869, DE 100 36 627.9-33, DE 100 28 397.7,

DE 101 10 530.4, DE 101 11 014.6, PCT/EP 00/10516,  
EP 01 102 674.7, PACT13, PACT17, PACT18, PACT22, PACT24,  
PACT25, PACT26US, PACT02, PACT04, PACT08, PACT10, PACT15,  
PACT18(a), PACT 27, PACT19. Diese sind hiermit zu Offenba-  
rungszwecken vollumfänglich eingegliedert.

VPUs bestehen grundsätzlich aus einer mehrdimensionalen homo-  
genen oder inhomogenen, flachen oder hierarchischen Anordnung  
(PA) von Zellen (PAEs), die beliebige Funktionen, i.b. logi-  
sche und/oder arithmetische Funktionen und/oder Speicherfunk-  
tionen und/oder Netzwerkfunktionen ausführen können. Den PAEs  
ist typisch eine Ladeeinheit (CT) zugeordnet, die die Funktion  
der PAEs durch Konfiguration und ggf. Rekonfiguration be-  
stimmt.

Das Verfahren basiert auf einem abstrakten parallelen Maschi-  
nenmodell, das neben dem endlichen Automaten auch imperative  
Problemspezifikationen integriert und eine effiziente algo-  
rithmische Ableitung einer Implementierung auf unterschiedli-  
che Technologien ermöglicht.

Folgende Compilerklassen sind nach dem Stand der Technik be-  
kannt:

Klassische Compiler, die häufig Stack-Maschinen-Code generie-  
ren und für sehr einfache Prozessoren geeignet waren, die im  
Wesentlichen als normale Sequenzer ausgestaltet sind. (vgl.  
N.Wirth, Compilerbau, Teubner Verlag).

Vektorisierender Compiler bauen weitgehend linearen Code, der  
auf spezielle Vektorrechner oder stark gepipelinte Prozessoren  
abgestimmt ist. Ursprünglich waren diese Compiler für Vektor-  
rechner wie CRAY verfügbar. Moderne Prozessoren wie Pentium  
benötigen aufgrund der langen Pipelinestruktur ähnliche Ver-  
fahren. Da die einzelnen Rechenschritte vektorisiert (gepipe-

lined) ablaufen ist der Code sehr viel effizienter. Allerdings bereitet der bedingte Sprung Probleme für die Pipeline. Daher ist eine Sprungvorhersage sinnvoll, die ein Sprungziel annimmt. Ist die Annahme falsch, muss jedoch die gesamte Verarbeitungspipeline gelöscht werden. Mit anderen Worten ist jeder Sprung für diese Compiler problematisch, eine Parallelverarbeitung im eigentlichen Sinn ist nicht gegeben. Sprungvorhersagen und ähnliche Mechanismen erfordern einen erheblichen Zusatzaufwand an Hardware.

Grobgranulare parallele Compiler existieren im eigentlichen Sinne kaum, die Parallelität wird typischerweise durch den Programmierer oder das Betriebssystem markiert und verwaltet, also beispielsweise bei MMP-Computersystemen wie verschiedenen IBM Architekturen, ASCI Red, etc. zumeist auf Thread-Ebene durchgeführt. Ein Thread ist ein weitgehend unabhängiger Programmblock oder gar ein anderes Programm. Threads sind daher grobgranular einfach zu parallelisieren. Die Synchronisation und Datenkonsistenz ist vom Programmierer bzw. dem Betriebssystem sicherzustellen. Dies ist aufwendig zu programmieren und erfordert einen wesentlichen Anteil der Rechenleistung eines Parallelrechner. Zudem ist durch diese grobe Parallelisierung nur ein Bruchteil der eigentlich möglichen Parallelität tatsächlich nutzbar.

Feingranulare parallele (z.B. VLIW) Compiler versuchen die Parallelität feingranular in VLIW Rechenwerke abzubilden, die mehrere Rechenoperationen in einem Takt parallel ausführen können aber einen gemeinsamen Registersatz besitzen. Ein wesentliches Problem stellt dieser limitierte Registersatz dar, da er die Daten für sämtliche Rechenoperationen bereitstellen muss. Zudem erschweren Datenabhängigkeiten und inkonsistente Lese/Schreiboperationen (LOAD/STORE) die Parallelisierung.

Rekonfigurierbare Prozessoren weisen eine große Anzahl an unabhängigen Rechenwerken auf, die typisch in einem Feld angeordnet sind. Diese sind typisch nicht durch einen gemeinsamen Registersatz, sondern durch Busses miteinander verbunden. Dadurch lassen sich einerseits leicht Vektorrechenwerke aufbauen, andererseits können auch einfach parallele Operationen durchgeführt werden. Durch die Busverbindungen werden entgegen der herkömmlichen Registerkonzepte Datenabhängigkeiten aufgelöst.

Die Aufgabe der vorliegenden Erfindung besteht darin, Neues für die gewerbliche Anwendung bereitzustellen.

Die Lösung dieser Aufgabe wird in unabhängiger Form beansprucht. Bevorzugte Ausführungsformen finden sich in den Unteransprüchen.

Es wird also vorgeschlagen, für einen Compiler für rekonfigurierbare Prozessoren die Konzepte von vektorisierenden Compilern und parallelisierenden (z.B. VLIW) Compilern zugleich anzuwenden und somit auf feingranularer Ebene zu Vektorisieren und parallelisieren.

Ein wesentlicher Vorteil besteht darin, dass der Compiler nicht auf eine fest vorgegebene Hardwarestruktur abbilden muss, sondern die Hardwarestruktur mit dem erfindungsgemäßen Verfahren so konfiguriert werden kann, dass sie optimal für die Abbildung des jeweiligen compilierten Algorithmus geeignet ist.

## **2. Beschreibung**

Als Grundlage zur Abarbeitung praktisch jeder Methodik zur Spezifizierung von Algorithmen wird der endliche Automat genutzt. Die Struktur eines endlichen Automaten ist in Figur 1

abgebildet. Ein einfacher endlicher Automat zerfällt in ein kombinatorisches Netz und eine Registerstufe zum Zwischenspeichern von Daten zwischen den einzelnen Datenverarbeitungszyklen.

Ein endlicher Automat führt eine Anzahl rein kombinatorischer (also z.B. logischer und/oder arithmetischer) Datenmanipulationen aus, um danach einen stabilen Zustand zu erreichen, der in einem Register(satz) repräsentiert wird. Basierend auf diesem stabilen Zustand wird entschieden, welcher nächste Zustand im nächsten Verarbeitungsschritt erreicht werden soll und somit auch, welche kombinatorischen Datenmanipulationen im nächsten Schritt durchgeführt werden sollen.

Beispielsweise repräsentiert ein Prozessor oder Sequenzer einen endlichen Automaten. In einem ersten Verarbeitungsschritt kann eine Subtraktion von datendurchgeführt werden. Das Ergebnis wird gespeichert. Im nächsten Schritt kann basierend auf dem Ergebnis der Subtraktion ein Sprung durchgeführt werden, der je nach Vorzeichen des Ergebnisses in eine andere Weiterverarbeitung führt.

Der endliche Automat ermöglicht die Abbildung komplexer Algorithmen auf beliebige sequentielle Maschinen, wie in Figur 2 abgebildet. Der dargestellte komplexe endliche Automat besteht aus einem komplexen kombinatorischen Netz, einem Speicher zum Ablegen von Daten und einem Adressgenerator zum Adressieren der Daten im Speicher.

Nun kann jedes beliebige sequentielle Programm grundlegend als endlicher Automat interpretiert werden, wobei aber zumeist ein sehr großes kombinatorisches Netz entsteht. Bei der Programmierung klassischer "von Neumann"-Architekturen - also bei allen CPUs - werden daher die kombinatorischen Operationen in eine Folge von jeweils einzelnen einfachen, fest vorgegebenen

Operationen (OpCodes) auf CPU-interne Register zerlegt. Durch diese Zerlegung entstehen Zustände zur Steuerung der in eine Folge zerlegten kombinatorischen Operation, die aber innerhalb der ursprünglichen kombinatorischen Operation per se nicht vorhanden sind, bzw. nicht benötigt werden. Daher sind jedoch die abzuarbeitenden Zustände einer von Neumann Maschinen grundsätzlich von den algorithmischen Zuständen eines kombinatorischen Netzes, also den Registern endlicher Automaten zu unterscheiden.

Es wurde nun erkannt, dass die VPU-Technologie (wie sie im Wesentlichen durch einige oder alle der Schriften PACT01, PACT02, PACT03, PACT04, PACT05, PACT08, PACT10, PACT13, PACT17, PACT18, PACT22, PACT24 definiert ist, die durch Bezugnahme vollumfänglich eingegliedert sind) im Gegensatz zu den starren OpCodes von CPUs ermöglicht, komplexe Instruktionen entsprechend eines abzubildenden Algorithmus wie in flexiblen Konfigurationen hineinzukompilieren.

## 2.1 Arbeitsweise des Compilers

Besonders vorteilhaft ist bei der Arbeitsweise des Compilers, wenn die komplexen Instruktionen derart generiert werden, daß diese möglichst lange in der PAE-Matrix ohne Rekonfiguration ausgeführt wird.

Der Compiler generiert weiterhin den endlichen Automaten bevorzugt aus dem imperativen Quelltext derart, daß er der jeweiligen PAE-Matrix besonders gut angepasst ist, also solche Operationen darin vorgesehen werden, die die typisch grobgranularen Logikkreise (ALUs etc.), gegebenenfalls auch vorhandene feingranulare Elemente (FPGA-Zellen in der VPU, statemachines etc.) besonders effizient nutzen.



Der compilererzeugte endliche Automat wird dann in Konfigurationen zerlegt.

Das Abarbeiten (Interpretieren) des endlichen Automaten geschieht auf einer VPU derart, daß die generierten Konfigurationen successive auf die PAE-Matrix abgebildet werden und die Arbeitsdaten und/oder Zustände, die zwischen den Konfigurationen zu übertragen sind, in Speicher abgelegt werden. Dazu kann das aus PACT04 bekannte Verfahren bzw. die entsprechende Architektur verwendet werden. Dieser Speicher wird vom Compiler bestimmt beziehungsweise vorgesehen. Es repräsentiert eine Konfiguration dabei eine Mehrzahl von Instruktionen; eine Konfiguration bestimmt zugleich für eine Vielzahl von Takten die Arbeitsweise der PAE-Matrix, während dieser Takte wird eine Vielzahl von Daten in der Matrix verarbeitet; diese stammen aus einer VPU externen Quelle und/oder einem internen Speicher und werden an eine externe Quelle und/oder einen internen Speicher geschrieben. Die internen Speicher ersetzen dabei den Registersatz einer CPU nach dem Stand der Technik derart, daß z.B. ein Register durch einen Speicher repräsentiert wird, wobei nicht ein Datenwort je Register gespeichert wird, sondern ein gesamter Datensatz je Speicher.

Wesentlich kann auch sein, daß die Daten und/oder Zustände der Verarbeitung einer ablaufenden Konfiguration compilerbestimmt in die Speicher abgelegt werden und somit der nächsten ablaufenden Konfiguration zur Verfügung stehen.

Ein bedeutender Unterschied zu Compilern, die auf Instruktionsbasis parallelisieren, besteht somit darin, daß das Verfahren die PAE-Matrix derart konfiguriert und rekonfiguriert, dass eine konfigurierte Folge von kombinatorischen Netzen auf einer VPU emuliert wird, während herkömmliche Compiler geladene Folgen von Instruktionen (OpCodes) kombinieren, wobei eine

Instruktion als ein kombinatorisches Netz betrachtet werden kann.

## 2.2 Ausführungsbeispiel WHILE-Sprache

Im Folgenden soll die Funktionsweise des Compilers anhand einer einfachen Sprache beispielhaft verdeutlicht werden. Dabei wird von einer Sprache ausgegangen, die in ihren Grundlagen bereits bekannt ist, wobei in einer bekannten Veröffentlichung [Referenz "Doktorarbeit Armin Nückel"] jedoch lediglich die Abbildung einer Funktion auf ein statisches kombinatorisches Netz beschrieben wird, während mit der Erfindung nun die Abbildung auf Konfigurationen erfolgt, die dann in einer zeitlichen Reihenfolge entsprechend des Algorithmus und der sich während der Verarbeitung ergebenden Zustände auf die PAE-Matrix abgebildet werden.

Die Programmiersprache geht davon aus, dass neben einfachen logischen und/oder arithmetischen Verknüpfungen ein Befehl "WHILE" existiert, der mit folgender Syntax definiert ist:

WHILE...

Mögliche Konstrukte sind damit:

- Anweisung
- Folge von Anweisungen
- Schleife

Eine Anweisung oder eine Folge von Anweisungen ist durch das beschriebene Compiler-Verfahren auf ein kombinatorisches Netz abbildbar.

**Figur 3a** zeigt ein kombinatorisches Netz mit den dazugehörigen Variablen. Dabei kann sich der Inhalt ein und derselben Variable (z.B. x1) von einer Stufe (0301) des Netzes zur nächsten (0302) ändern.

Diese Veränderung ist beispielsweise für die Zuweisung

$x1 := x1 + 1$

in **Figur 3b** dargestellt.

Zur Adressierung zum Lesen der Operanden und zum Speichern der Ergebnisse können nun Adressgeneratoren mit dem kombinatorischen Netz der Zuweisung synchronisiert werden. Mit jeder verarbeiteten Variable werden entsprechende neue Adressen für Operanden und Ergebnisse generiert (**Figur 3c**). Die Art des Adressgenerators ist prinzipiell beliebig und hängt von den Adressierungsschematas der compilierten Applikation ab. Für Operanden und Ergebnisse können gemeinsame, kombinierte oder vollständig unabhängige Adressgeneratoren implementiert werden.

Typischerweise werden bei der Datenverarbeitung wie im vorliegenden Datenverarbeitungsmodell eine Mehrzahl von Daten innerhalb einer bestimmten Konfiguration der PAEs verarbeitet. Bevorzugt ist der Compiler daher für die in vielen, wenn nicht den meisten Anwendungen möglichen einfachen FIFO-Modus ausgelegt, der zumindest für die Datenspeicher anwendbar ist, die innerhalb dieser Beschreibung zum Speichern von Daten und Zuständen der Datenverarbeitung (quasi als Ersatz eines gewöhnlichen Registersatzes herkömmlicher CPUs) dienen. Mit anderen Worten dienen Speicher der temporären Speicherung von Variablen zwischen den Konfigurationen. Auch hier ist eine Konfiguration ähnlich einer Instruktion eines normalen Prozessors und die Speicher (insbesondere eine Mehrzahl) sind vergleichbar mit dem Registersatz eines normalen Prozessors.

### 2.2.3 Folgen von Anweisungen

Eine Folge der beispielhaften Zuweisung läßt sich wie folgt generieren (**Figur 4a**):

```
x1 := 0;  
WHILE TRUE DO  
    x1 := x1 + 1;
```

Diese Folge läßt sich nunmehr mittels einer Zuweisung gemäß 2.2.1 und Adressgeneratoren für Operanden und Ergebnisse abbilden.

### Endliche Folgen

Der Vollständigkeit halber soll eine besondere Ausgestaltung von Folgen abseits der definierten Konstrukte der WHILE Sprache diskutiert werden. Eine endliche Folge der beispielhaften Zuweisung läßt sich wie folgt generieren:

```
FOR i:=1 TO 10  
    x1 := x1 + 1;
```

Eine derartige Folge läßt sich durch zwei Arten implementieren:

- a) Durch Generierung eines Addierers zur Berechnung von  $i$  entsprechend des WHILE-Konstruktes (siehe 2.2.4) und eines weiteren Addierers zur Berechnung von  $x1$ . Die Folge wird als Schleife abgebildet und iterativ berechnet (Figur 5a).
- b) Durch Auswalzen der Schleife, wodurch die Berechnung von  $i$  als Funktion entfällt. Die Berechnung von  $x1$  wird  $i$ -mal instantiiert und als Pipeline aufgebaut, wodurch  $i$  nacheinander geschaltete Addierer entstehen (Figur 5b).

### 2.2.4 Bedingungen

Bedingungen lassen sich mittels WHILE ausdrücken. Beispielsweise:

```
x1 := 0;  
WHILE x1 < 10 DO  
    x1 := x1 + 1;
```

Die Abbildung generiert eine zusätzliche PAE zur Verarbeitung des Vergleiches. Das Vergleichsergebnis wird durch ein Statussignal repräsentiert (vgl. Trigger in PACT08), das von den

die Anweisung verarbeitenden PAEs und den Adressgeneratoren ausgewertet wird.

Die resultierende Abbildung ist in **Figur 4b** dargestellt.

Durch die Auswertung der Bedingung (hier WHILE, generell und einleuchtenderweise auch andere Anweisungen wie IF; CASE realisierbar) wird ein Status generiert, der der nachfolgenden Datenverarbeitung zur Verfügung gestellt werden kann (DE 197 04 728.9) und/oder an die CT oder eine lokale Ladesteuerung (DE 196 54 846.2) gesendet werden kann, die daraus Information über den weiteren Programmfluß und evtl. anstehende Rekonfigurationen ableitet.

#### 2.2.5 Grundlegendes Verfahren

Entsprechend den vorherigen Verfahren wird jedes Programm in ein System abgebildet, das wie folgt aufgebaut ist:

1. Speicher für Operanden (vgl. Register einer CPU)
2. Speicher für Ergebnisse (vgl. Register einer CPU)
3. Netzwerk aus a) Zuweisungen und/oder b) Vergleichen-Anweisungen, also Bedingungen wie z.B. IF, CASE, Schleifen (WHILE, FOR, REPEAT)
4. Optionalen Adressgenerator(en) zur Ansteuerung der Speicher nach 1 und 2.

#### 2.2.6 Umgang mit Zuständen

Es wird nun für die Zwecke des beschriebenen Compilers zwischen algorithmisch relevanten und irrelevanten Zuständen unterschieden. Zustände werden in der VPU-Technologie für gewöhnlich durch Statussignale (z.B. Trigger in PACT08) und/oder Handshakes (z.B. RDY/ACK in PACT02) dargestellt. Generell können Zustände (v.a. in anderen Technologien, wie FPGAs, DPGAs, Chameleon-Bausteinen, Morphics, etc.) durch beliebige Signale,

Signalbündel und/oder Register dargestellt werden. Das offenbarte Compilerverfahren kann auch auf solche angelegt werden, obwohl wesentliche Teile der Beschreibung bevorzugt auf die VPU fokussieren.

Relevante Zustände sind innerhalb des Algorithmus notwendig um dessen korrekte Funktion zu beschreiben. Sie sind für den Algorithmus wesentlich.

Irrelevante Zustände entstehen durch die verwendete Hardware und/oder die gewählte Abbildung oder aus anderen sekundären Gründen. Sie sind damit für die Abbildung (also die Hardware) wesentlich.

Lediglich die relevanten Zustände müssen mit den Daten erhalten werden. Daher werden diese zusammen mit den Daten in den Speichern abgelegt, da sie entweder als Ergebnis der Verarbeitung mit den Daten auftraten oder als Operanden mit den Daten für den nächsten Verarbeitungszyklus notwendig sind.

Irrelevante Zustände sind dagegen nur örtlich und/oder zeitlich lokal notwendig und müssen daher nicht gespeichert werden.

#### Beispiel:

- a) Die Zustandsinformation eines Vergleichs ist für die weitere Verarbeitung der Daten relevant, da dieser die auszuführenden Funktionen bestimmt.
- b) Angenommen, ein sequentieller Dividierer entsteht beispielsweise durch Abbildung eines Divisionsbefehles auf eine Hardware, die nur die sequentielle Division unterstützt. Dadurch entsteht ein Zustand, der den Rechenschritt innerhalb der Division kennzeichnet. Dieser Zustand ist irrelevant, da für den Algorithmus nur das Ergebnis (also die ausgeführte Division) erforderlich ist. In diesem Fall werden also lediglich das Ergebnis und die Zeitinformation (also die Verfügbarkeit) benötigt. Der Compiler unterschei-

det solche relevante und irrelevante Zustände bevorzugt voneinander.

Die Zeitinformation ist beispielsweise in der VPU-Technologie nach PACT01, 02, 13 durch das RDY/ACK Handshake erhältlich. Hierzu ist jedoch besonders anzumerken, dass das Handshake ebenfalls keine relevanten Zustand darstellt, da es lediglich die Gültigkeit der Daten signalisiert, wodurch sich wiederum die verbleibende relevante Information auf die Existenz gültiger Daten reduziert.

#### 2.2.7 Umgang mit Zeit

In vielen Programmiersprachen, besonders in sequentiellen wie z.B. C, wird eine exakte zeitliche Reihenfolge implizit durch die Sprache vorgegeben; bei sequentiellen Programmiersprachen geschieht dies beispielsweise durch die Reihenfolge der einzelnen Anweisungen. Sofern dies durch die Programmiersprache und/oder den Algorithmus erforderlich ist, wird das Compilerverfahren so ausgeführt, dass sich die Zeitinformation auf Synchronisationsmodelle wie RDY/ACK und/oder REQ/ACK oder ein Time-Stamp-Verfahren nach DE 101 10 530.4 abbilden lässt.

Mit anderen Worten wird die implizite Zeitinformation von sequentiellen Sprachen in ein Handshake Protokoll derart abgebildet, dass das Handshake Protokoll (RDY/ACK-Protokoll) die Zeitinformation überträgt und insbesondere die Reihenfolge der Zuweisungen garantiert.

Beispielsweise wird die nachfolgende for-Schleife nur dann durchlaufen und iteriert, wenn die Variable inputstream je Durchlauf mit einem RDY quittiert ist. Bleibt RDY aus, wird der Schleifendurchlauf bis zum Eintreffen von RDY angehalten.

```
while TRUE
```

```
s := 0
for i:= 1 to 3
    s := s + inputstream;
```

Die Eigenschaft der sequentiellen Sprachen, nur von der Befehlsverarbeitung gesteuert zu werden, wird somit bei der Compilierung mit dem Datenflußprinzip, die Verarbeitung durch den Datenstrom, bzw. die Existenz von Daten zu steuern verbunden. Mit anderen Worten wird ein Befehl und/oder eine Anweisung (z.B. `s := s + inputstream;`) nur verarbeitet, wenn die Operation ausgeführt werden kann und die Daten verfügbar sind.

Bemerkenswert ist, daß dieses Verfahren gewöhnlicherweise zu keiner Änderung der Syntax oder Semantik einer Hochsprache führt. Es kann also vorhandener Hochsprachencode durch Neucompilierung problemfrei zur Ausführung auf einer VPU gebracht werden.

#### 2.2.8 Laden und Speichern von Daten

Für die Grundlagen der Load/Store Operationen ist folgendes beachtlich.

Die nachfolgenden Adressierungsarten werden unterschiedlich behandelt:

1. externe Adressierung, also die Datentransfers mit externen Baugruppen
2. interne Adressierung, also die Datentransfers zwischen PAEs, i.b. zwischen RAM-PAEs und ALU-PAEs

Desweiteren kann die zeitliche Entkopplung der Datenverarbeitung und dem Laden und Speichern der Daten besondere Beachtung finden.



Bustransfers werden in interne und externe Transfers zerlegt.

bt1) Externe Lesezugriffe (Load Operation) werden separiert, in einer möglichen Ausführung auch bevorzugt in eine separate Konfiguration übersetzt. Die Daten werden von einem externen Speicher in einen internen transferiert.

bt2) Interne Zugriffe werden mit der Datenverarbeitung gekoppelt, d.h. die internen Speicher (Register Operation) werden für die Datenverarbeitung gelesen, bzw. beschrieben.

bt3) Externe Schreibzugriffe (Store Operation) werden separiert, in einer bevorzugten möglichen Ausführung auch in eine separate Konfiguration übersetzt. Die Daten werden von einem internen Speicher in einen externen transferiert.

Wesentlich ist, dass die bt1, bt2, bt3 - also das Laden der Daten (Load), das Verarbeiten der Daten (Datenverarbeitung und bt2) und das Schreiben der Daten (bt3) - in unterschiedliche Konfigurationen übersetzt werden können und diese ggf. zu einem unterschiedlichen Zeitpunkt ausgeführt werden.

Das Verfahren soll an dem nachfolgenden Beispiel verdeutlicht werden:

```
function example (a, b : integer) -> x : integer
for i:= 1 to 100
  for j:= 1 to 100
    x[i] := a[i] * b[j]
```

Die Funktion kann vom Compiler in drei Teile, bzw. Konfigurationen (subconfig) transformiert:  
example#dload: Lädt die Daten von extern (Speicher, Peripherie, etc.) und schreibt diese in interne Speicher. Interne Speicher sind mit r# und dem Namen der ursprünglichen Variable gekennzeichnet.

example#process: Entspricht der eigentlichen Datenverarbeitung. Diese liest die Daten aus internem Operanden und schreibt die Ergebnisse wieder in interne Speicher.  
example#dstore: schreibt die Ergebnisse aus dem internen Speicher nach extern (Speicher, Peripherie, etc.).

```
function example# (a, b : integer) -> x : integer
subconfig example#dload
for i := 1 to 100
  r#a[i] := a[i]
for j := 1 to 100
  r#b[j] := b[j]

subconfig example#process
for i := 1 to 100
  for j:= 1 to 100
    r#x[i] := r#a[i] * r#b[j]

subconfig example#dstore
for i := 1 to 100
  x[i] := r#x[i]
```

Ein wesentlicher Effekt des Verfahrens ist, dass anstatt  $i*j = 100 * 100 = 10.000$  externe Zugriffe nur  $i+j = 100 + 100 = 200$  externe Zugriffe zum Lesen der Operanden ausgeführt werden. Diese Zugriffe sind zudem noch vollkommen linear, was die Transfergeschwindigkeit bei modernen Bussystemen (Burst) und/oder Speichern (SDRAM, DDRAM, RAMBUS, etc) erheblich beschleunigt.

Die internen Speicherzugriffe erfolgen parallel, da den Operanden unterschiedliche Speicher zugewiesen wurden.

Zum Schreiben der Ergebnisse sind  $i = 100$  externe Zugriffe notwendig, die ebenfalls wieder linear mit maximaler Performance erfolgen können.

Wenn die Anzahl der Datentransfers vorab nicht bekannt ist (z.B. WHILE-Schleifen) oder sehr groß ist, kann ein Verfahren verwendet werden, das bei Bedarf durch Unterprogrammaufrufe die Operanden nachlädt bzw. die Ergebnisse nach extern schreibt. Dazu können in einer bevorzugten Ausführung (auch) die Zustände der FIFOs abgefragt werden: 'empty' wenn das FIFO leer ist, sowie 'full' wenn das FIFO voll ist. Entsprechend der Zustände reagiert der Programmfluß. Zu bemerken ist, dass bestimmte Variablen (z.B.  $ai$ ,  $bi$ ,  $xi$ ) global definiert sind. Zu Performanceoptimierung kann ein Scheduler entsprechend der bereits beschriebenen Verfahren die Konfigurationen `example#dloada`, `example#dloadb` bereits vor dem Aufruf von `example#process` bereits ausführen, sodass bereits Daten vorgeladen sind. Ebenso kann `example#dstore(n)` nach der Terminierung von `example#process` noch aufgerufen werden um  $r\#x$  zu leeren.

```
subconfig example#dloada(n)
```

```
  while !full(r#a) AND ai<=n
    r#a[ai] := a[ai]
    ai++
```

```
subconfig example#dloadb(n)
```

```
  while !full(r#b) AND bi<=n
    r#b[bi] := b[bi]
    bi++
```

```
subconfig example#dstore(n)
```

```
  while !empty(r#x) AND xi<=n
    x[xi] := r#x[xi]
    xi++
```

```

subconfig example#process
for i := 1 to n
  for j:= 1 to m
    if empty(r#a) then example#dloada(n)
    if empty(r#b) then example#dloadb(m)
    if full(r#x) then example#dstore(n)

    r#x[i] := r#a[i] * r#b[j]
  bj := 1

```

Die Unterprogrammaufrufe und das Verwalten der globalen Variablen sind für rekonfigurierbare Architekturen vergleichsweise aufwendig. Daher kann in einer bevorzugten Ausführung die nachfolgende Optimierung durchgeführt werden, in welcher sämtliche Konfigurationen weitgehend unabhängig ablaufen und nach vollständiger Abarbeitung terminieren (terminate). Da die Daten b[j] mehrfach erforderlich sind, muß example#dloadb entsprechend mehrfach durchlaufen werden. Dazu werden beispielsweise zwei Alternativen dargestellt:

Alternative 1: example#dloadb terminiert nach jedem Durchlauf und wird von example#process für jeden Neustart neu konfiguriert.

Alternative 2: example#dloadb läuft unendlich und wird von example#process terminiert.

Während 'idle' ist eine Konfiguration untätig (wartend).

```

subconfig example#dloada(n)
  for i:= 1 to n
    while full(r#a)
      idle
    r#a[i] := a[i]
  terminate

subconfig example#dloadb(n)

```

```
while 1 // ALTERNATIVE 2
  for i:= 1 to n
    while full(r#b)
      idle
    r#b[i] := a[i]
  terminate

subconfig example#dstore(n)
  for i:= 1 to n
    while empty(r#b)
      idle
    x[i] := r#x[i]
  terminate

subconfig example#process
  for i := 1 to n
    for j:= 1 to m
      while empty(r#a) or empty(r#b) or full(r#x)
        idle
      r#x[i] := r#a[i] * r#b[j]
config example#dloadb(n) // ALTERNATIVE 1
terminate example#dloadb(n) // ALTERNATIVE 2
  terminate
```

Zur Vermeidung von Wartezyklen können Konfigurationen auch terminiert werden, sobald sie ihre Aufgabe temporär nicht weiter erfüllen können. Die entsprechende Konfiguration wird von dem rekonfigurierbaren Baustein entfernt, verbleibt jedoch im Scheduler. Hierzu wird im Folgenden der Befehl 'reenter' verwendet. Die relevanten Variablen werden vor der Terminierung gesichert und bei der wiederholten Konfiguration wiederhergestellt:

```
subconfig example#dloada(n)
```

```
for ai:= 1 to n
  if full(r#a) reenter
  r#a[ai] := a[ai]
terminate

subconfig example#dloadb(n)
  while 1 // ALTERNATIVE 2
    for bi:= 1 to n
      if full(r#b) reenter
      r#b[bi] := a[bi]
    terminate

subconfig example#dstore(n)
  for xi:= 1 to n
    if empty(r#b) reenter
    x[xi] := r#x[xi]
  terminate

subconfig example#process
  for i := 1 to n
    for j:= 1 to m
      if empty(r#a) or empty(r#b) or full(r#x) reenter
      r#x[i] := r#a[i] * r#b[j]
      config example#dloadb(n) // ALTERNATIVE 1
      terminate example#dloadb(n) // ALTERNATIVE 2
    terminate
```

### 2.3 Makros

Komplexere Funktionen einer Hochsprache, wie z.B. Schleifen, werden typisch durch Makros realisiert. Die Makros werden dabei vom Compiler vorgegeben und zur Übersetzungszeit instantiiert. (vgl. Figur 4).

Die Makros sind entweder aus einfachen Sprachkonstrukten der Hochsprache oder auf Assemblerlevel aufgebaut. Makros können parametrisiert sein, um eine einfache Adaption an den beschriebenen Algorithmus zu ermöglichen. (vgl. Figur 5, 0502). Die Makros sind auch vorliegend einzugliedern.

#### 2.4 Feedback Loops und Register

Innerhalb der Abbildung eines Algorithmus in ein kombinatorisches Netz können unverzögerte Rückkopplungen entstehen, die unkontrolliert schwingen.

In praktisch implementierten VPU-Technologien gemäß PACT02 wird dies durch einen Aufbau der PAE verhindert, bei welchem mindestens ein Register zur Entkopplung, etwa fest in den PAEs, definiert ist.

Generell sind unverzögerte Rückkopplungen durch Graphenanalyse des entstandenen kombinatorischen Netzes feststellbar. In die Datenpfade, in denen eine unverzögerte Rückkopplung besteht, werden daraufhin gezielt Register zur Entkopplung eingefügt. Der Compiler kann somit Register- beziehungsweise Speichermitel verwalten.

Durch die Verwendung von Handshake-Protokollen (z.B. RDY/ACK gem. 2.2.7) ist die korrekte Funktion der Berechnung auch durch das Einfügen von Registern sichergestellt.

#### 2.5 Prozssormodell / Time Domain Multiplexing (TDM)

Grundsätzlich besitzt jede praktisch realisierte PAE-Matrix lediglich eine endliche Größe. Daher muß im folgenden Schritt eine Partitionierung des Algorithmus nach 2.2.5 Abs. 4 a/b in eine Mehrzahl von Konfigurationen durchgeführt werden, die nacheinander in die PAE-Matrix konfiguriert werden. Ziel ist typisch, möglichst viele Datenpakete in dem Netzwerk zu berechnen, ohne umkonfigurieren zu müssen.

Zwischen den Konfigurationen wird eine Zwischenspeicherung vorgenommen, wobei der Zwischenspeicher - ähnlich eines Registers bei CPUs - die Daten zwischen den einzelnen sequentiell ausgeführten Konfigurationen speichert.

Somit wird durch das Rekonfigurieren von Konfigurationen, der Datenverarbeitung in der PAE-Matrix und der Zwischenspeicherung in den Speichern ein sequentielles Prozessormodell aufgebaut.

Mit anderen Worten wird in der VPU-Technologie durch die beschriebene Compilierung nicht ein OpCode sequentiell ausgeführt, sondern komplexe Konfigurationen. Während bei CPUs ein Opcode typischerweise ein Datenwort bearbeitet, werden in der VPU-Technologie eine Mehrzahl von Datenworten (ein Datenpaket) von einer Konfiguration bearbeitet. Dadurch steigt die Effizienz der rekonfigurierbaren Architektur durch ein besseres Verhältnis zwischen Rekonfigurationsaufwand und Datenverarbeitung.

In der VPU-Technologie wird zugleich anstatt eines Registers ein Speicher verwendet, da nicht Datenworte, sondern Datenpakete zwischen den Konfigurationen bearbeitet werden.

Dieser Speicher kann als Random-Access Memory, Stack, FIFO oder als beliebige andere Speicherarchitektur ausgestaltet sein, wobei typischerweise mit einem FIFO die beste und am einfachsten zu realisierende Möglichkeit gegeben ist.

Daten werden nunmehr durch die PAE-Matrix, entsprechend des konfigurierten Algorithmus, bearbeitet und in einem oder mehreren Speichern gespeichert. Die PAE-Matrix wird nach der Bearbeitung einer Menge von Daten umkonfiguriert und die neue Konfiguration entnimmt die Zwischenergebnisse aus dem/den Speicher(n) und setzt die Ausführung des Programmes fort. Da-



bei können durchaus auch neue Daten von externen Speichern und/oder der Peripherie zusätzlich in die Berechnung einfließen, ebenso können Ergebnisse an externen Speichern und/oder der Peripherie geschrieben werden.

Mit anderen Worten ist der typische Ablauf einer Datenverarbeitung das Auslesen von internen RAMs, das Verarbeiten der Daten in der Matrix und das Schreiben von Daten in die internen Speicher, wobei zur Datenverarbeitung auch beliebige externe Quellen oder Ziele für Datentransfers zusätzlich oder anstelle der internen Speicher verwendet werden können.

Während "sequenzen" bei CPUs als das Neuladen eines OpCodes definiert ist, wird nach dem Vorstehenden "sequenzen" von VPU's also als das (Re)konfigurieren von Konfigurationen definiert. Dies bedeutet allerdings nicht, daß nicht unter bestimmten Bedingungen Teile des Feldes als Sequenzer im herkömmlichen Sinne betrieben werden könnten.

Die Information, wann und/oder wie gesequenzte wird, d.h. welche nächste Konfiguration konfiguriert werden soll, ist durch verschiedene Informationen darstellbar, die einzeln oder kombiniert verwendet werden können. Z. B. sind folgende Strategien zur Ableitung der Information allein und/oder in Kombination bzw. alternativ sinnvoll:

- a) durch den Compiler zur Übersetzungszeit definiert;
- b) durch das Event-Netzwerk definiert  
(Trigger, DE 197 04 728.9),  
wobei das Eventnetzwerk interne und/oder externe Zustände repräsentieren kann;
- c) durch den Füllgrad der Speicher  
(Trigger, DE 197 04 728.9, DE 196 54 846.2-53).

#### 2.5.1 Einfluß des TDM auf das Prozessormodell

Die Partitionierung des Algorithmus bestimmt entscheidend die relevanten Zustände, die in den Speichern zwischen den verschiedenen Konfigurationen abgelegt werden. Sofern ein Zustand nur innerhalb einer Konfiguration relevant ist (lokal relevanter Zustand), ist es nicht notwendig, diesen zu speichern, was vom Compilierverfahren bevorzugt berücksichtigt wird.

Zu Zwecken des Debuggings des auszuführenden Programmes kann es aber sinnvoll sein, diese Zustände dennoch zu speichern, um dem Debugger einen Zugriff auf diese zu ermöglichen. Auf die DE 101 42 904.5 wird verwiesen; diese ist hiermit vollumfänglich zu Offenbarungszwecken eingegliedert.

Weiterhin können zusätzlich Zustände relevant werden, wenn ein Taskswitch-Mechanismus (z.B. durch ein Betriebssystem oder Interruptquellen) verwendet wird und aktuelle ausgeführte Konfigurationen unterbrochen werden, andere Konfigurationen geladen werden und/oder zu einem späteren Zeitpunkt die abgebrochene Konfiguration fortgesetzt werden soll. Eine detailliertere Beschreibung folgt im nachfolgenden Abschnitt.

Ein einfaches Beispiel soll ein Unterscheidungsmerkmal für lokal relevante Zustände aufzeigen:

- a) Eine Verzweigung des Types "if () then ... else ..." paßt vollständig in eine einzige Konfiguration, d.h. beide Datenpfade (Zweige) sind gemeinsam vollständig innerhalb der Konfiguration abgebildet. Der sich beim Vergleich ergebende Zustand ist relevant, jedoch lokal, da er in den nachfolgenden Konfigurationen nicht mehr benötigt wird.
- b) Dieselbe Verzweigung ist zu groß, um vollständig in eine einzige Konfiguration zu passen. Mehrere Konfigurationen sind notwendig, um die vollständigen Datenpfade abzubilden. In diesem Fall ist der Zustand global relevant und muß gespeichert und den jeweiligen Daten zugeordnet werden, da die nachfolgenden Konfigurationen bei der Weiterverarbei-

tung der Daten den jeweiligen Zustand des Vergleichs benötigen.

## 2.6 Task-Switching

Einen zusätzlichen Einfluß auf die Betrachtung und den Umgang mit Zuständen hat der mögliche Einsatz eines Betriebssystems. Betriebssysteme verwenden beispielsweise Task-Scheduler zum Verwalten mehrerer Aufgaben (Tasks), um ein Multitasking zur Verfügung zu stellen.

Task-Scheduler brechen Tasks zu einem bestimmten Zeitpunkt ab, starten andere Tasks und kehren nach deren Abarbeitung zur Weiterbearbeitung des abgebrochenen Tasks zurück.

Sofern sichergestellt ist, daß eine Konfiguration - die hier der Abarbeitung eines Tasks entsprechen kann - nur nach der kompletten Abarbeitung - d.h. wenn alle innerhalb dieses Konfigurationszyklusses zu bearbeitende Daten und Zustände gespeichert sind - terminiert, können lokal relevante Zustände ungespeichert bleiben. Dieses Verfahren, also das komplette Abarbeiten einer Konfiguration und der nachfolgende Taskswitch ist die bevorzugte Methode für den Betrieb von rekonfigurierbaren Prozessoren und entspricht im Wesentlichen dem Ablauf in einem normalen Prozessor, der auch zunächst die aktuell bearbeiteten Instruktionen abarbeitet und dann den Task wechselt.

Für manche Anwendungen ist jedoch eine besonders kurze Reaktion auf eine Taskwechselsanforderung erforderlich, z.B. in Realtime-Anwendungen. Hier kann es sinnvoll sein Konfigurationen vor deren kompletter Abarbeitung abzubrechen.

Sofern der Task-Scheduler Konfigurationen vor deren vollständiger Abarbeitung abbricht, müssen lokale Zustände und/oder Daten gespeichert werden. Weiterhin ist dies von Vorteil, wenn die Abarbeitungszeit einer Konfiguration nicht vorhergesagt werden kann. In Verbindung mit dem bekannten Halteproblem und dem Risiko, daß eine Konfiguration (z.B. durch einen Fehler)

gar nicht terminiert, erscheint dies weiterhin sinnvoll, um damit einen Deadlock des gesamten Systems zu verhindern. Daher sind vorliegend, unter Berücksichtigung von Taskwechseln, relevante Zustände auch als solche anzusehen, die für einen Taskwechsel und ein erneutes korrektes Aufsetzen der Datenverarbeitung notwendig sind.

Bei einem Taskswitch ist somit der Speicher für Ergebnisse und ggf. auch der Speicher für die Operanden zu sichern und zu einem späteren Zeitpunkt, also bei der Rückkehr zu diesem Task, wieder herzustellen. Dies kann vergleichbar zu den PUSH/POP Befehlen und Verfahren nach dem Stand der Technik erfolgen. Weiterhin ist der Zustand der Datenverarbeitung zu sichern, also der Zeiger auf die zuletzt vollständig bearbeiteten Operanden. Es sei hier besonders auf PACT18 verwiesen.

Abhängig von der Optimierung des Taskswitches gibt es beispielsweise zwei Möglichkeiten:

- a) Die abgebrochene Konfiguration wird neu konfiguriert und nur die Operanden werden geladen. Die Datenverarbeitung beginnt so von neuem, als ob die Bearbeitung der Konfiguration noch gar nicht begonnen wurde. Mit anderen Worten werden einfach alle Datenberechnungen von vorne an ausgeführt, wobei ggf. Berechnungen bereits zuvor durchgeführt wurden. Diese Möglichkeit ist einfach, aber nicht effizient.
- b) Die abgebrochene Konfiguration wird neu konfiguriert, wobei die Operanden und die bereits berechneten Ergebnisse in die jeweiligen Speicher geladen werden. Die Datenverarbeitung wird bei den Operanden fortgesetzt, die nicht mehr vollständig berechnet wurden. Dieses Verfahren ist sehr viel effizienter, setzt aber voraus, daß ggf. zusätzliche Zustände, die während der Verarbeitung der Konfiguration entstehen, relevant werden, etwa wenn zumindest ein Zeiger auf die zuletzt vollständig verrechneten Operanden gesichert

werden muss, damit bei deren Nachfolgern nach erfolgter neuer Konfiguration neu aufgesetzt werden kann.

## 2.7 Kontext Switch

Eine besonders bevorzugte Variante zur Verwaltung von relevanten Daten wird durch den nachfolgend beschriebenen Kontext Switch zur Verfügung gestellt. Bei Task-Wechseln und/oder bei der Ausführung von Konfigurationen und deren Wechsel (siehe beispielsweise Patentanmeldung DE 102 06 653.1, die zu Offenbarungszwecken vollumfänglich eingegliedert ist) kann es erforderlich sein, Daten oder Zustände, die typischerweise nicht zusammen mit den Arbeitsdaten in die Speicher abgelegt werden, da sie beispielsweise lediglich einen Endwert markieren, für eine nachfolgende Konfiguration zusichern.

Der erfindungsgemäß bevorzugt implementierte Kontext Switch wird derart durchgeführt, dass eine erste Konfiguration entfernt wird und die zu sichernden Daten in entsprechenden Speichern (REG) (Speicher, Register, Zähler, etc) verbleiben.

Dann kann eine zweite Konfiguration geladen werden, diese verbindet die REG in geeigneter Weise und definierter Reihenfolge mit einem oder mehreren globalen Speicher(n).

Die Konfiguration kann beispielsweise Adressgeneratoren verwenden, um auf den/die globalen Speicher zuzugreifen. Es ist also nicht erforderlich, vorab jeden einzelnen Speicherplatz durch den Compiler festlegen zu lassen und/oder auf als Speicher ausgestaltete REG zuzugreifen.

Entsprechend der konfigurierten Verbindung zwischen den REG werden die Inhalte der REG in einer definierten Reihenfolge in den globalen Speicher geschrieben, wobei die jeweiligen Adressen von Adressgeneratoren vorgegeben werden. Der Adressgenerator generiert die Adressen für den/die globalen Speicher(n) derart, dass die beschriebenen Speicherbereiche (PUSHAREA) der

entfernten ersten Konfiguration eindeutig zugeordnet werden können.

Es werden somit bevorzugt für unterschiedliche Konfigurationen unterschiedliche Adressenräume vorgesehen. Die Konfiguration entspricht dabei einem PUSH gewöhnlicher Prozessoren.

Danach verwenden andere Konfigurationen die Ressourcen.

Nun soll die erste Konfiguration wieder gestartet werden. Zuvor wird eine dritte Konfiguration gestartet, die die REG der ersten Konfiguration in einer definierten Reihenfolge miteinander verbindet.

Die Konfiguration kann wiederum beispielsweise Adressgeneratoren verwenden um auf den oder die globalen Speichern zuzugreifen und/oder um auf als Speicher ausgestaltete REG zuzugreifen.

Ein Adressgenerator generiert dabei Adressen bevorzugt derart, dass ein korrekter Zugriff auf die der ersten Konfiguration zugeordnete PUSHAREA erfolgt. Die generierten Adressen und die konfigurierte Reihenfolge der REG sind derart, dass die Daten der REG in der ursprünglichen Ordnung aus den Speichern in die REG geschrieben werden. Die Konfiguration entspricht einem POP gewöhnlicher Prozessoren.

Nun wird die erste Konfiguration wieder gestartet.

Zusammengefaßt wird ein Kontext Switch bevorzugt derart durchgeführt, dass durch das Laden besonderer Konfigurationen, die ähnlich von PUSH/POP bekannter Prozessorarchitekturen arbeiten, die zu sichernden Daten mit einem globalen Speicher ausgetauscht werden. Dieser Datenaustausch über globale Speicher mittels von Push/Pop-Austauschkonfigurationen wird als besonders relevant angesehen.

Die Funktion soll in einem Beispiel verdeutlicht werden:  
Eine Funktion addiert 2 Zahlenreihen, die Länge der Reihen ist zur Übersetzungszeit nicht bekannt, sondern erst zur Laufzeit.

```
proc example
  while i<length do
    x[i] = a[i] + b[i]
    i = i + 1
```

Die Funktion wird nun während ihrer Ausführung unterbrochen, beispielsweise durch einen Task-Switch, oder weil der für x vorgesehene Speicher voll ist. a,b,x befinden sich zu diesem Zeitpunkt erfindungsgemäß in Speichern. i und ggf. length müssen jedoch gesichert werden.

Dazu wird die Konfiguration example terminiert, wobei die Registerinhalte erhalten bleiben und eine Konfiguration push gestartet, die i und length aus den Registern liest und in einen Speicher schreibt.

```
proc push
  mem[<push_adr_example>] = i
  push_adr_example++
  mem[<push_adr_example>] = length
```

Nach der Ausführung wird push terminiert und die Registerinhalte können gelöscht werden.

Andere Konfigurationen werden ausgeführt. Nach einiger Zeit wird die Konfiguration example wieder gestartet.

Zuvor wird eine Konfiguration pop gestartet, die die Registerinhalte wieder aus dem Speicher liest.

```
proc pop
  i = mem[<push_adr_example>]
```

```
push_adr_example++  
length = mem[<push_adr_example>]
```

Nach der Ausführung wird pop terminiert und die Registerinhalte bleiben bestehen. Die Konfiguration example wird wieder gestartet.

## 2.8 Algorithmische Optimierung

Durch das beschriebene Übersetzungsverfahren werden Kontrollstrukturen von algorithmischen Strukturen getrennt. Beispielsweise zerfällt eine Schleife in einen Rumpf (WHILE) und eine algorithmische Struktur (Anweisungen).

Die algorithmischen Strukturen lassen sich nunmehr bevorzugt optional durch ein zusätzliches, der Trennung nachgeschaltetes Werkzeug optimieren.

Beispielsweise kann eine nachgeschaltete Algebra-Software die programmierten Algorithmen optimieren und minimieren. Derartige Tools sind z.B. unter Bezeichnungen wie AXIOM, MARBLE, etc. bekannt. Durch die Minimierung kann eine schnellere Ausführung des Algorithmusses und/oder ein erheblich verringerter Platzbedarf erreicht werden.

Das Ergebnis der Optimierung wird danach wieder in den Compiler geführt und entsprechend weiterverarbeitet.

Es soll zudem angemerkt sein, dass moderne Compiler (-Frontends) bereits eine Anzahl von Optimierungen für Algorithmen (auch z.T. algebraische) implementiert haben, die selbstverständlich im Rahmen des hier beschriebenen Verfahrens weiterhin nutzbar sind.

Es soll ausdrücklich erwähnt sein, dass die beschriebenen Verfahren, insbesondere jedoch die Abschnitte 2.2.7 "Umgang mit Zeit" und 2.3 "Makros" auch auf Compiler nach PACT20 angewendet werden können. PACT20 wird diesbezüglich zu Offenbarungszwecken vollumfänglich in diese Patentanmeldung einbezogen.



### 3. Anwendbarkeit für Prozessoren nach dem Stand der Technik, insbesondere mit VLIW-Architektur

Es soll besonders angemerkt werden, daß anstatt einer PAE-Matrix auch eine Anordnung von arithmetisch logischen Einheiten nach dem Stand der Technik (ALUs), wie beispielsweise in VLIW-Prozessoren üblich, und/oder eine Anordnung von kompletten Prozessoren, wie beispielsweise in Multiprozessorsystemen üblich, verwendet werden kann. Ein Sonderfall stellt dabei die Verwendung einer einzelnen ALU dar, sodaß das Verfahren auch für normale CPUs verwendbar ist.

In der Dissertation [Referenz Dissertation Armin Nückel] wurde ein Verfahren entwickelt, das die Übersetzung der WHILE-Sprache in semantisch korrekte endliche Automaten ermöglicht. Darüber hinaus kann ein endlicher Automat als "Unterprogramm" verwendet werden und umgekehrt. Dadurch entsteht die Möglichkeit, eine Konfiguration auf unterschiedliche Implementierungstechnologien abzubilden, wie z.B. CPUs; symmetrische Multiprozessoren; FPGAs; ASICs; VPU's.

Insbesondere ist es möglich, Teilen einer Applikation die jeweils optimal geeignete Hardware zuzuordnen bzw. eine jeweilige Eignung zu bestimmen und anhand der mehr oder weniger guten Eignung die optimale Hardware zuzuordnen. Dabei sind bevorzugt auch temporäre Ressourcenverteilungen und -reservierungen erfaßbar. Mit anderen Worten würde beispielsweise eine Datenflußstruktur einer Datenflußarchitektur zugeordnet werden, während eine sequentielle Struktur auf einen Sequenzer abgebildet wird, sofern diese vorhanden und/oder verfügbar sind.

Die entstehenden Problemstellungen der Ressourcenzuweisungen für die einzelnen Algorithmen können z.B. durch einen "Job As-

signment"-Algorithmus zur Verwaltung der Zuordnung gelöst werden.

#### 4. Implementierung

Die Implementierung eines erfindungsgemäßen Compilers soll von einer "normalen" sequentiellen Programmiersprache ausgehen, also z.B. C oder Pascal. Diese Sprachen weisen die Eigenschaft auf, dass durch ihren sequentiellen Charakter eine zeitliche Abfolge implizit und künstlich durch die Sprachdefinition an sich generiert wird.

Beispiel A:

```
Zeile 1:  i++  
Zeile 2:  a = i * b  
Zeile 3:  x = p - a  
Zeile 4:  j = i * i
```

Durch die Sprachdefinition ist fest vorgegeben, dass Zeile 1 vor Zeile 2 vor Zeile 3 vor Zeile 4 ausgeführt wird. Allerdings könnte Zeile 4 auch direkt nach Zeile 1 ausgeführt werden und somit parallel zu Zeile 2 und 3 bearbeitet werden.

Mit anderen Worten werden durch sequentielle Sprachen weitere künstliche und nicht algorithmisch bedingte Zustände eingebaut. Wichtig ist lediglich die korrekte zeitliche Abfolge der Berechnungen in Beispiel A. Zeile 4 darf erst berechnet werden, wenn *i* korrekt definiert ist, also nach der Abarbeitung von Zeile 1. Auch Zeile 2 darf erst nach der korrekten Definition von *i* (also nach der Abarbeitung von Zeile 1) verarbeitet werden. Zeile 3 benötigt die Ergebnisse von Zeile 2 (die Variable *a*) und darf daher erst nach derer korrekten Definition berechnet werden. Somit ergibt sich eine Datenabhängigkeit aber keine besonderen Zustände.

Anhand der Datenabhängigkeiten der Variable *a* in Zeile 2 und 3 (Zeile 3 verwendet *a* als Operand, *a* ist das Ergebnis von Zeile

2) kann automatisch durch den Compiler folgende Transformation zur Repräsentation der Parallelisier- bzw. Vektorisierbarkeit (ParVec-Transformation) durchgeführt werden:

```
Zeile 2:    VEC{a = i * b;  
Zeile 3:      x = p - a}
```

VEC bedeutet, dass jeder durch ';' getrennte Ausdruck nacheinander abgearbeitet wird, wobei die Ausdrücke innerhalb der geschweiften Klammern grundsätzlich gepipelinet werden können. Bevorzugt müssen sämtliche Berechnungen am Ende von VEC{} durchgeführt und abgeschlossen sein, damit die Datenverarbeitung hinter VEC fortgesetzt wird.

Besser wird in einer internen Repräsentation der Datenstrukturen im Compiler die beiden Berechnungen als ein Vektor markiert:

```
VEC{a=i*b; x=p-q}
```

Zeile 4 ergibt einen einfachen Vektor:

```
VEC{j = i*i}
```

Da sich Zeile 4 gleichzeitig zu Zeile 2 und 3 berechnen lässt kann die Parallelität folgendermassen ausgedrückt werden:

```
PAR{{VEC{a=i*b; x=p-a};VEC{j=i*i}}}
```

PAR bedeutet, dass jeder durch '{...}' getrennte Ausdruck zeitgleich abgearbeitet werden kann. Bevorzugt müssen sämtliche Berechnungen am Ende von PAR{} durchgeführt und abgeschlossen sein, damit die Datenverarbeitung hinter PAR fortgesetzt wird.

Wird Zeile 1 mit einbezogen, ergibt sich:

```
VEC{i++; PAR{{VEC{a=i*b; x=p-a}}{{VEC{j=i*i}}}}
```

Da  $\text{VEC}\{j=i*i\}$  ein Vektor mit nur einem Element darstellt, kann auch wie folgt geschrieben werden:

```
VEC{i++; PAR{{VEC{a=i*b; x=p-a}}{j=i*i}}}
```

Ein weiteres Beispiel zeigt einen echten Zustand.

Beispiel B:

```
Zeile 1:  i++
Zeile 2:  a = i * b
Zeile 3:  if a < 100 {
Zeile 4:      x = p - a
Zeile 5:  } else {
Zeile 6:      j = i * i }
```

Jetzt kann Zeile 6 nur noch nach der Berechnung von Zeile 2 und Zeile 3 ausgeführt werden. Die Berechnung von Zeile 4 und 6 findet alternativ statt. Also ist der Zustand von Zeile 3 für die weitere Datenverarbeitung relevant (relevanter Zustand).

Bedingte Zustände können bei einer Transformation durch IF ausgedrückt werden:

```
Zeile 1-2:  VEC{i++; a=i*b}
Zeile 3:    IF{{a<100}{zeile4}{zeile6}}
Zeile 4:    VEC{x=p-a}
Zeile 6:    VEC{j=i*i}
```

Zusammengefaßt ergibt das

```
VEC{i++; a=i*b; IF{{a<100}{VEC{x=p-a}}{VEC{j=i*i}}}}
```

Weitere relevante Zustände werden durch Schleifen erzeugt:

Beispiel C:

Zeile 1:     for (i = 1, i < 100, i++)

Zeile 2:         a = a \* i

Zeile 3:     q = p / a

Zeile 3 darf erst ausgeführt werden, nachdem die Schleife terminiert ist. Also bestehen bei bedingten Sprüngen relevante Zustände.

Eine erste Transformation der Schleife ergibt:

Zeile 1:         i=1;

Zeile 2: loop: if i >= 100 then exit

Zeile 3:         a = a \* i

Zeile 4:         i++

Zeile 5:         jump loop

Zeile 6: exit: q = p / a

Zeile 3 und 4 können parallel berechnet werden, da Zeile 4 nicht vom Ergebnis von Zeile 3 abhängt:

PAR({a=a\*i}{i++})

Zeile 5 ergibt einen Vektor mit dem generierten PAR, da erst nach vollständiger Berechnung der Werte wieder in die Schleife gesprungen werden darf (hier liegt also eine zeitliche Abhängigkeit vor).

VEC{PAR({a=a\*i}{i++}); jump loop}

Somit ergibt sich für die Bedingung:

loop: IF({i>=100}{jump exit}{VEC{PAR({a=a\*i}{i++}); jump loop}})

Die Zeile 1 ist ein Vektor mit der Bedingung, da diese vor der Bedingung ausgeführt werden muss (IF verwendet i als Operand, i ist das Ergebnis von Zeile 1).

Zeile 6 ist wiederum ein Vektor mit der Bedingung, da a als Operand verwendet wird und a das Ergebnis der Bedingung ist.

Somit ergibt sich (in übersichtlicher Schreibweise):

```
VEC{
    i++;
    loop: IF{
        {i>=100}
        {jump exit}
        {VEC{
            PAR{
                {a=a*i}
                {i++}
            };
            jump loop
        }
    };
    exit: q=p/a
}
```

Die Inhalte von VEC{} und PAR{} können als rein kombinatorische Netze betrachtet werden.

Bevorzugt wird VEC und PAR als Petri-Netz ausgestaltet, um wie bevorzugt die Weiterverarbeitung nach kompletter Verarbeitung der jeweiligen Inhalte zu steuern.

Durch die mögliche Betrachtung von VEC und PAR als rein kombinatorisches Netz entsteht die Notwendigkeit den Schleifenzustand zu sichern. D.h. in diesem Fall ist es tatsächlich notwendig einen endlichen Automaten zu schaffen.

Die Anweisung REG{} speichert dazu Variablen in einem Register. Somit entsteht durch die Verwendung der kombinatorischen Netze VEC und PAC in Verbindung mit dem Register REG ein endlicher Automat, der exakt entsprechend des Algorithmus aufgebaut ist:

```
VEC{
    i++;
    loop: IF{
        {i>=100}
        {jump exit}
        {VEC{
            PAR{
                {a=a*i}
                {i++}
            };
            REG(a;i)
            jump loop
        }
    }
};
exit: q=p/a
}
```

Es soll besonders darauf hingewiesen werden, dass in der VPU Technologie des Anmelders (vgl. PACT21) Ein- und/oder Ausgangsregister an den PAEs vorgesehen sind und die zeitliche Korrektheit und die Verfügbarkeit von Daten durch ein integriertes Handshake-Protokoll (RDY/ACK) sichergestellt ist. Insoweit wird die Forderung bevorzugt beim Verlassen von VEC{} oder PAR{} deren interne Datenverarbeitung abgeschlossen zu haben automatisch für alle nachfolgend verwendeten Variablen erfüllt (wäre die Datenverarbeitung nicht beendet, würden nachfolgende Berechnungsschritte auf die Beendigung und das

Eintreffen der Daten warten). Durch die integrierten Register sind auch schwingende Rückkopplungen ausgeschlossen.

Insoweit ist nachfolgender Term für diese Technologie korrekt:

```
VEC{PAR({a=a*i}{i++}); jump loop}
```

Für andere Technologien, die die o.g. Ausgestaltungen nicht oder nur teilweise aufweisen, sollte der Term folgendermassen formuliert werden:

```
VEC{PAR({a=a*i}{i++}); REG{a;i}; jump loop}
```

Es soll darauf hingewiesen werden, dass diese Form auf jeden Fall auch in der VPU-Technologie des Anmelders zu einer korrekten und optimalen Abbildung des Algorithmus auf den rekonfigurierbaren Prozessor führt.

REG kann innerhalb der kombinatorischen Netze VEC und PAR verwendet werden. Streng betrachtet verlieren dadurch VEC und PAR die Eigenschaft der kombinatorischen Netze. Abstrakt kann jedoch REG als ein komplexes Element (REG-Element) eines kombinatorischen Netzes betrachtet werden, dem eine eigene Abarbeitungszeit zugrunde liegt. Die Bearbeitung der nachfolgenden Elemente wird von der Beendigung der Berechnung des REG-Elementes abhängig gemacht.

In dem Bewusstsein dieser begrifflichen Ungenauigkeit wird eine Verwendung von REG innerhalb von VEC und PAR im Weiteren zugelassen und ist insbesondere auch notwendig.

Wie bereits vorstehend erwähnt, ist die Verwendung von REG typischerweise innerhalb einer Konfiguration einer VPU des Anmelders nicht erforderlich, sondern explizit immer nur dann, wenn die Berechnungsergebnisse einer Konfiguration abgespeichert werden, sodass REG ein diesem Anwendungsfall tatsächlich dem expliziten Register eines endlichen Automaten entspricht.



Neben der Synthese von endlichen Automaten für Schleifen, sind insbesondere in einem weiteren Fall endliche Automaten erforderlich:

Ist ein Algorithmus zu groß, um komplett innerhalb der PAEs eines rekonfigurierbaren Prozessors abgearbeitet zu werden, muß er in mehrere Teilalgorithmen zerlegt werden. Jeder Teilalgorithmus stellt eine Konfiguration für den rekonfigurierbaren Prozessor dar. Nacheinander, also sequentiell, werden die Teilalgorithmen auf den Prozessor konfiguriert, wobei die Ergebnisse der jeweils vorhergehenden Konfiguration(en) für die jeweils neue Konfiguration als Operanden dienen.

Mit anderen Worten entsteht durch die Rekonfiguration ein endlicher Automat, der zu einem Zeitpunkt  $t$  Daten bearbeitet und speichert und zu einem Zeitpunkt  $t+1$ , möglicherweise nach einer Konfiguration, die gespeicherten Daten ggf. anders verarbeitet und wieder speichert. Wesentlich ist, dass  $t$  nicht im klassischen Sinn durch Takte oder Befehle definiert wird, sondern durch Konfigurationen. Hierzu sein besonders die Präsentation Prozessormodell (PACT, Oktober 2000, San Jose) referenziert.

Mit noch anderen Worten besteht eine Konfiguration aus einem kombinatorischen Netz aus VEC und/oder PAR, dessen Ergebnisse gespeichert werden (REG), um in der nächsten Konfiguration weiterverwendet zu werden:

Konfiguration 1:    VEC{Operands;{VEC|PAR};REG{Results1}}

Konfiguration 2:    VEC{Results1;{VEC|PAR};REG{Results2}}

...

Zum einfacheren Verständnis haben die obigen Beispiels und Beschreibungen die Konstrukte VEC, PAR und REG in der Hochsprache eingeführt und diese dadurch strukturiert. Typischerweise

und bevorzugt wird diese Strukturierung erst aber auf der Ebene der Zwischensprache (siehe Principles of Compiler Design (Red Dragon), Aho, Sethi, Ullmann) eingeführt.

Es soll besonders darauf hingewiesen werden, dass die Strukturierung von Algorithmen mit VEC, PAR und REG typischerweise vollkommen automatisch durch den Compiler durch Methoden wie z.B. Graphenanalyse durchführbar ist.

Insbesondere ist es aber auch denkbar und teilweise von Vorteil dem Programmierer selbst die Strukturierungsmöglichkeit in der Hochsprache dadurch zu ermöglichen, dass VEC, PAR und REG wie oben aufgezeigt direkt in der Hochsprache beschreibbar sind.

### Generierung

Die automatische Erstellung von VEC, PAR und REG kann auf unterschiedlichen Ebenen einen Compilierungsvorganges durchgeführt werden. Die zunächst einleuchtendste ist während eines Präprozessor-Durchlaufes auf Basis des Source-Codes wie in den vorigen Beispielen beschrieben. Für die weitere Compilierung ist danach allerdings ein speziell angepasster Compiler erforderlich.

Ein weiterer Aspekt ist, dass Compiler zumeist automatische Optimierungen von Code vornehmen (z.B. in Schleifen). Eine effiziente Zerlegung des Codes ist daher erst nach den Optimierungsläufen sinnvoll, insbesondere wenn Compiler (wie z.B. SUIF, Universität Stanford) bereits den Code für Parallelisierung und/oder Vectorisierung hin optimieren.

Die daher besonders bevorzugte Methode ist die Einbindung der Analysen in das Backend eines Compilers. Das Backend übersetzt eine compilerinterne Datenstruktur auf die Befehle eines Zielprozessors.

Als compilerinterne Datenstrukturen werden zumeist Zeigerstrukturen wie DAGs/GAGs, Trees oder 3-Adress-Codes verwendet (siehe Principles of Compiler Design (Red Dragon), Aho, Sethi, Ullmann). Teilweise werden auch Stack-Machine-Codes verwendet (siehe Compiler selbstgeschneidert, C'T 1986 1-5). Da die Datenformate prinzipiell äquivalent sind und ineinander transformiert werden können, setzt die erfindungsgemäß bevorzugte Methode auf der Weiterverarbeitung von Graphen, wie bevorzugt Trees, auf.

Datenabhängigkeiten und mögliche Parallelitäten entsprechend dem vorstehend beschriebenen Verfahren sind innerhalb von Trees einfach auf Basis der Struktur automatisch zu erkennen. Hierzu können beispielsweise bekannte und etablierte Verfahren der Graphenanalyse eingesetzt werden. Alternativ oder optional kann durch entsprechend adaptierte Parsingmethoden ein Algorithmus auf Datenabhängigkeiten, Schleifen, Sprünge etc. hin untersucht werden. Dabei kann ein Verfahren ähnlich dem der Auswertung von Ausdrücken in Compilern verwendet werden.

### **Abbildung**

Die weitere Transformation des Algorithmus ist nunmehr stark von der Zielarchitektur abhängig. Beispielsweise bietet die Prozessorarchitektur des Anmelders (VPU, XPP) automatische Datensynchronisation in Hardware. Das bedeutet, dass die korrekte zeitliche Abfolge von Datenabhängigkeiten automatisch in der Hardware gehandhabt wird. Andere Architekturen benötigen zum Teil zusätzlich die Synthese geeigneter Zustandsmaschinen für die Steuerung des Datentransfers.

Besonders interessant ist die Handhabung bedingter Sprünge. Beispielsweise stellt die Prozessorarchitektur des Anmelders

mehrere Mechanismen zu deren Abbildung und Ausführung zur Verfügung:

1. Rekonfiguration des Prozessors oder Teilen des Prozessors durch eine übergeordnete Konfigurationseinheit (vgl. Patentanmeldung(en) PACT01, 04, 05, 10, 13, 17)
2. Auswalzen der Funktion in das Array aus PAEs (vgl. Patentanmeldung PACT08), dabei werden z.B. beide möglichen Zweige eines Vergleiches zugleich auf das Array abgebildet.
3. Wave Rekonfiguration nach Patentanmeldung(en) PACT08, 13, 17), dabei wird den unterschiedlich zu bearbeitenden Daten ein Token mitgegeben, das die jeweils gültige Konfiguration wählt.

Es soll erwähnt sein, dass der Mechanismus 1 der allgemein typisch anzuwendende Fall ist. Der Mechanismus 2 ist bereits bei den meisten Technologien sehr aufwendig oder gar nicht implementierbar und der Fall 3 ist bislang nur aus der VPU-Technologie des Anmelders bekannt.

Die jeweils zu wählende Ausführungsmethode hängt von der Komplexität des Algorithmus, dem erforderlichen Datendurchsatz (Performance) und der exakten Ausgestaltung des Zielprozessors ab (z.B. Anzahl der PAEs).

Beispiele:

Ein einfacher Vergleich soll folgendes Berechnen:

```
if i < 0 then a=a*(-i) else a=a*i
```

Eine Rekonfiguration des Prozessors (Mechanismus 1) je nach Ergebnis des Vergleichs scheint wenig sinnvoll zu sein.

Das Auswalzen beider Zweige in das Array (Mechanismus 2) ist Grundsätzlich möglich. Je nach Ergebnis des Vergleichs werden entweder die  $a=a*(-i)$  oder  $a=a*i$  berechnenden PAEs angesteuert (vgl. PACT08).

Besonders platzeffizient ist das Überlagern der beiden Berechnungen (Mechanismus 3), wodurch nach dem Vergleich unabhängig vom Ergebnis dieselbe(n) PAEs die Daten weiterverarbeiten, die

Daten aber mit einem Token versehen sind, das sodann in Abhängigkeit vom Vergleich lokal in den jeweils nachfolgenden die Daten verarbeitenden PAEs entweder die Funktion  $a=a*(-i)$  oder  $a=a*i$  auswählt. (vgl. PACT08, 13, 17).

Nach Mechanismus 1 entsteht ein global relevanter Zustand, da die komplette folgende Konfiguration davon abhängig ist.

Nach Mechanismus 2 und 3 entstehen nur ein lokal relevanter Zustand, da dieser über die Berechnung hinaus - die vollständig implementiert ist - nicht mehr benötigt wird.

Mit anderen Worten kann die lokale oder globale Relevanz von Zuständen auch von der gewählten Abbildung auf die Prozessorarchitektur abhängen.

Ein Zustand der über eine Konfiguration hinaus und somit über das kombinatorische Netz des eine Konfiguration repräsentierenden endlichen Automaten hinaus relevant ist (also von nachfolgenden endlichen Automaten benötigt wird), kann grundsätzlich als global betrachtet werden. Es soll nochmals auf die verwendete diffuse Terminologie des Begriffes kombinatorisches Netz hingewiesen werden.

#### **Befehlsmodell des entstandenen Prozessors**

Entsprechend der vorliegenden Erfindung entsteht ein Prozessormodell für rekonfigurierbare Prozessoren, das alle wesentlichen Befehle umfaßt:

Arithmetisch/logische Befehle werden direkt in das kombinatorische Netz abgebildet.

Sprünge (Jump/Call) werden entweder direkt in das kombinatorische Netz ausgewalzt oder durch Rekonfiguration realisiert.

Bedingung und Kontrollflußbefehle (if, etc) werden entweder im kombinatorischen Netz vollständig aufgelöst und bearbeitet oder an eine übergeordnete Konfigurationseinheit weitergeleitet, die sodann entsprechend des entstandenen Status eine Rekonfiguration durchführt.

Load/Store-Operationen werden bevorzugt in separate Konfigurationen abgebildet und durch Adressgeneratoren ähnlich den bekannten DMA's realisiert, die internen Speicher (REG{}) mittels Adressgeneratoren in externe Speicher schreiben oder diese von externen Speichern und/oder Peripherie laden. Sie können aber auch zusammen mit der datenverarbeitenden Konfiguration konfiguriert sein und arbeiten.

Register-Move-Operationen werden im kombinatorischen Netz durch Busse zwischen den internen Speichern (REG{}) realisiert.

Push/Pop-Operationen werden durch separate Konfigurationen realisiert, die ggf. bestimmte interne Register im kombinatorischen Netz und/oder die internen Speicher (REG{}) mittels Adressgeneratoren in externe Speicher schreiben oder aus externen Speichern lesen und die bevorzugt vor oder nach den eigentlichen datenverarbeitenden Konfigurationen ausgeführt werden.

## **5. Beschreibung der Figuren**

Die nachfolgenden Figuren zeigen Implementierungs- und Ausstattungsbeispiele des Compilers.

**Figur 1a** zeigt den Aufbau eines gewöhnlichen endlichen Automaten, bei welchem ein kombinatorisches Netz (0101) mit einem Register (0102) verknüpft ist. Daten können direkt an 0101 (0103) und 0102 (0104) geführt werden. Durch eine Rückkopplung (0105) des Registers auf das kombinatorische Netz ist die Verarbeitung eines Zustandes in Abhängigkeit des/der vorhergehen-

den Zustände möglich. Die Verarbeitungsergebnisse werden durch 0106 dargestellt.

**Figur 1b** zeigt eine Repräsentation des endlichen Automaten durch eine rekonfigurierbare Architektur nach PACT01 und PACT04 (PACT04 Fig. 12-15). Das kombinatorischen Netz aus Figur 1a (0101) wird durch eine Anordnung von PAEs 0107 ersetzt (0101b). Das Register (0102) wird durch einen Speicher (0102b) ausgeführt, der mehrere Zyklen speichern kann. Die Rückkopplung gemäß 0105 erfolgt durch 0105b. Die Eingänge (0103b bzw. 0104b) sind äquivalent 0103 bzw 0104. Der direkte Zugriff auf 0102b kann ggf. durch einen Bus durch das Array 0101b realisiert werden. Der Ausgang 0106b ist wiederum äquivalent 0106.

**Figur 2** zeigt die Abbildung eines endlichen Automaten auf eine rekonfigurierbare Architektur. 0201(x) repräsentieren das kombinatorische Netz (das entsprechend Figur 1b als PAEs ausgestaltet sein kann). Es existieren ein oder mehrere Speicher für Operanden (0202) und ein oder mehrere Speicher für Ergebnisse (0203). Zusätzliche Daten Ein-/Ausgänge gem. 0103b, 0104b, 0106b) sind der Einfachheit halber nicht dargestellt. Den Speichern zugeordnet ist jeweils ein Adressgenerator (0204, 0205).

Die Operanden- und Ergebnisspeicher (0202, 0203) sind physikalisch oder virtuell derart miteinander verkoppelt, daß beispielsweise die Ergebnisse einer Funktion bzw. einer Operation einer anderen als Operanden dienen können und/oder sowohl Ergebnisse als auch neu zugeführte Operanden einer Funktion einer anderen als Operanden dienen können. Eine derartige Kopplung kann beispielsweise durch Bussysteme hergestellt werden oder durch eine (Re)Konfiguration durch welche die Funktion und Vernetzung der Speicher mit den 0201 neu konfiguriert wird.

**Figur 3** zeigt verschiedene Aspekte zum Umgang mit Variablen.

In **Figur 3a** zeigen 0301, 0302, 0303 verschiedene Stufen der Berechnung. Diese Stufen können rein kombinatorisch oder auch über Register voneinander getrennt sein.  $f_1$ ,  $f_2$ ,  $f_3$  sind Funktionen,  $x_1$  ist eine Variable gemäß Patentbeschreibung.

**Figur 3b** zeigt die Verwendung einer Variablen  $x_1$  in der Funktion  $x_1 := x_1 + 1$ .

**Figur 3c** zeigt das Verhalten eines endlichen Automaten zur Berechnung von  $x_1 := x_1 + 1$  innerhalb einer Konfiguration. In der nächsten Konfiguration sind 0306 und 0304 zu vertauschen um einen vollständigen endlichen Automaten zu erhalten. 0305 repräsentiert die Adressgeneratoren für die Speicher 0304 und 0306.

**Figur 4** zeigt Implementierungen von Schleifen. Die schraffierten Module können durch Makros generiert werden (0420, 0421). 0421 kann auch durch Analyse des Graphen auf unverzögerte Rückkopplungen eingefügt werden.

**Figur 4a** zeigt die Implementierung einer einfachen Schleife der Art

```
WHILE TRUE DO
```

```
     $x_1 := x_1 + 1$ ;
```

Im Kern der Schleife liegt der Zähler +1 (0401). 0402 ist ein Multiplexer, der zu Beginn den Startwert von  $x_1$  (0403) auf 0401 führt und sodann bei jeder Iteration die Rückkopplung (0404a, 0404b) bewirkt. In die Rückkopplung ist ein Register (vgl. REG{ }) (0405) eingesetzt, um eine unverzögerte und damit unkontrollierte Rückkopplung des Ausgangs von 0401 auf dessen Eingang zu verhindern. 0405 wird mit dem Arbeitstakt der VPU getaktet und bestimmt damit die Anzahl der Iterationen pro Zeit. Der jeweilige Zählerstand wäre an 0404a oder 0404b abgreifbar. Je nach Definition der Hochsprache terminiert die Schleife jedoch nicht. Beispielsweise wäre in einer HDL (nach dem Stand der Technik (z.B. VHDL, Verilog) das Signal auf 0404 nutzbar, während es in einer sequentiellen Programmiersprache



(z.B. C) 0404 nicht nutzbar ist, da die Schleife nicht terminiert und somit keinen Exit-Wert liefert.

Der Multiplexer 0402 realisiert ein Makro, das aus dem Schleifenkonstrukt entstanden ist. Das Makro wird durch die Übersetzung von WHILE instantiiert.

Das Register 0405 ist entweder ebenfalls Teil des Makros oder wird entsprechend einer Graphenanalyse nach dem Stand der Technik exakt dann und dort eingefügt, wo eine unverzögerte Rückkopplung existiert, um so die Schwingneigung auszuschalten.

**Figur 4b** zeigt den Aufbau einer echten Schleife der Art  
WHILE x1 < 10 DO

```
    x1 := x1 + 1;
```

Der Aufbau entspricht im Kern der Figur 4a, weshalb dieselben Referenzen verwendet wurden.

Zusätzlich ist eine Schaltung dargestellt, die die Gültigkeit des Ergebnisses kontrolliert (0410) und das Signal von 0404a nur dann an die nachfolgenden Funktionen (0411) weiterleitet, wenn das Abbruchkriterium der Schleife erreicht ist. Das Abbruchkriterium wird durch den Vergleich  $x1 < 10$  festgestellt (Vergleichsstufe 0412). Als Ergebnis des Vergleiches wird das betreffende Statusflag (0413) an ein Multipliziermittel 0402 zur Steuerung der Schleife und die Funktionen 0411 zur Kontrolle der Ergebnisweiterführung geleitet. Das Statusflag 0413 kann beispielsweise durch Trigger gemäß DE 197 04 728.9 implementiert sein. Ebenfalls kann das Statusflagmittel 0413 an eine CT gesendet werden, die daraufhin die Terminierung der Schleife erkennt und eine Rekonfiguration durchführt.

**Figur 5a** zeigt die iterative Berechnung von  
FOR i:=1 TO 10

```
    x1 := x1 * x1;
```

Im wesentlichen entspricht die Grundfunktion Figur 4b, weshalb die Referenzen übernommen wurden. Der Funktionsblock 0501 berechnet die Multiplikation. Die FOR-Schleife wird durch eine

weitere Schleife entsprechend Figur 4b implementiert und ist lediglich durch Block 0503 angedeutet. Block 0503 liefert den Status des Vergleiches auf das Abbruchkriterium. Der Status wird direkt zur Ansteuerung der Iteration verwendet, wodurch das Mittel 0412 (dargestellt durch 0502) weitgehend entfällt.

**Figur 5b** zeigt das Auswalzen der Berechnung von

```
FOR i:=1 TO 10
```

```
    x1 := x1 * x1;
```

Da die Anzahl der Iterationen zur Übersetzungszeit exakt bekannt ist, kann die Berechnung in eine Folge von  $i$  Multiplizierern (0510) abgebildet werden.

**Figur 6** zeigt die Ausführung einer WHILE-Schleife gem. Figur 4b über mehrere Konfigurationen. Hier ist der Zustand der Schleife (0601) ein relevanter Zustand, da dieser die Funktion in den nachfolgenden Konfigurationen maßgeblich beeinflusst. Die Berechnung erstreckt sich über 4 Konfigurationen (0602, 0603, 0604, 0605). Zwischen den Konfigurationen werden die Daten in Speichern (vgl. REG{}) abgelegt (0606, 0607). 0607 ersetzt dabei ebenfalls 0405.

Als ein Rekonfigurationskriterium kann der Füllstand der Speicher dienen, angedeutet über 0606, 0607: Speicher voll/leer und/oder 0601, das den Abbruch der Schleife anzeigt. Mit anderen Worten werden durch den Füllstand der Speicher Trigger generiert (vgl. PACT01, PACT05, PACT08, PACT10), die an die Konfigurationseinheit gesendet werden und eine Rekonfiguration auslösen. Auch der Zustand der Schleife (0601) kann an die CT gesendet werden. Daraufhin kann die CT bei Erreichen des Abbruchkriteriums die nachfolgenden Algorithmen konfigurieren, bzw. ggf. zunächst die restlichen Teile der Schleife (0603, 0604, 0605) abarbeiten und danach die nachfolgenden Konfigurationen laden.

## 6. Parallelisierbarkeit

Figur 6 zeigt potentielle Grenzen der Parallelisierbarkeit auf.

Sofern die Berechnung der Operanden unabhängig von der Rückkopplung 0608 ist, kann die Schleife blockweise, d.h. jeweils durch Füllen der Speicher 0606/0607 berechnet werden. Damit wird ein hoher Grad an Parallelität erreicht.

Sofern die Berechnung eines Operanden abhängig von dem Ergebnis der vorherigen Berechnung ist, also eine Rückkopplung oder dergleichen 0608 in die Berechnung einfließt, wird das Verfahren ineffizienter, da jeweils nur ein Operand innerhalb der Schleife berechnet werden kann.

Ist der nutzbare ILP (Instruktionslevel Parallelismus) innerhalb der Schleife hoch und die Zeit für die Rekonfiguration nieder (vgl. PACT02, PACT04, PACT13, PACT17), kann eine auf PAEs ausgewählte Berechnung auf einer VPU weiterhin effizient sein.

Ist dies nicht der Fall, ist es sinnvoll, die Schleife auf eine sequentielle Architektur (vom PA separater Prozessor oder Implementierung innerhalb des PA entsprechend DE 196 51 075.9-53, DE 196 54 846.2-53 und insbesondere DE 199 26 538.0 (Fig. 5, 11, 16, 17, 23, 30, 31, 33)) abzubilden.

Die Analyse der Berechnungszeiten kann entweder im Compiler zur Übersetzungszeit gemäß dem nachfolgenden Abschnitt erfolgen und/oder empirisch zu der oder einer Laufzeit gemessen werden, um eine nachträgliche Optimierung herbeizuführen, was zu einem lernfähigen, insbesondere selbstlernenden Compiler führt.

Für die Erfindung sind Analyse- und Parallelisierungsverfahren von Bedeutung.

Verschiedene Verfahren nach dem Stand der Technik stehen für die Analyse und Durchführung der Parallelisierung zur Verfügung.

Ein bevorzugtes Verfahren soll im Folgenden beschrieben werden.

Abzubildende Funktionen werden durch Graphen dargestellt (vgl. PACT13; DE 199 26 538.0), wobei eine Applikation aus beliebig vielen unterschiedlichen Funktionen zusammengesetzt sein kann. Die Graphen werden auf die in ihnen enthaltene Parallelität untersucht, wobei vorab sämtliche Methoden der Optimierung zum Einsatz kommen können.

Beispielsweise sollen folgende Untersuchungen durchgeführt werden:

#### 6.0.1 ILP (Instruction Level Parallelism)

ILP drückt aus, welche Befehle zeitgleich ausgeführt werden können (vgl. PAR{{}). Eine derartige Analyse wird auf Basis der Betrachtung von Abhängigkeiten von Knoten in einem Graphen einfach möglich. Entsprechende Verfahren sind nach dem Stand der Technik und in der Mathematik per se hinreichend bekannt, es soll beispielsweise auf VLIW-Compiler und Synthesetools verwiesen werden.

Besondere Beachtung benötigen aber z. B. gegebenenfalls verschachtelte bedingte Ausführungen (IF), da eine korrekte Aussage der parallel ausführbaren Pfade oftmals kaum oder nicht zu treffen ist, da eine starke Abhängigkeit vom Werteraum der einzelnen Parameter besteht, der oftmals nicht oder nur unzureichend bekannt ist. Auch kann eine exakte Analyse derart viel Rechenzeit in Anspruch nehmen, daß sie nicht mehr sinnvoll durchführbar ist.

In derartigen Fällen kann beispielsweise die Analyse durch Hinweise vom Programmierer vereinfacht werden und/oder es kann anhand entsprechender Compilerschalter derart gearbeitet werden, daß im Zweifelsfall entweder von einer hohen Parallelisierbarkeit (ggf. unter Verschwendung von Ressourcen) oder von einer niederen Parallelisierbarkeit (ggf. unter Verschwendung von Performance) ausgegangen werden soll.

Ebenfalls kann in diesen Fällen eine empirische Analyse zur Laufzeit durchgeführt werden. Nach PACT10, PACT17 sind Verfahren bekannt, die zur Laufzeit die Erstellung von Statistiken über das Programmverhalten erlauben. Derart kann z. B. zunächst von einer maximalen Parallelisierbarkeit ausgegangen werden. Die einzelnen Pfade geben Meldungen an eine Statistikeinheit (z. B. implementiert in einer CT oder einer anderen Stufe, vgl. PACT10, PACT17, grundsätzlich können aber auch Einheiten nach PACT04 verwendet werden) über jeden Durchlauf zurück. Mittels statistischer Maßnahmen ist nunmehr auswertbar, welche Pfade tatsächlich parallel durchlaufen werden. Weiterhin ergibt sich die Möglichkeit, anhand der Daten zur Laufzeit zu bewerten, welche Pfade häufig oder selten oder nie parallel durchlaufen werden. Diese Art der Pfadnutzungsmeldung ist nicht zwingend, aber vorteilhaft.

Dementsprechend kann die Ausführung bei einem nächsten Programmaufruf optimiert werden. Daß dazu die Statistikinformation insbesondere nichtflüchtig wie auf eine Festplatte weggeschrieben werden kann, sei erwähnt. Aus PACT22, PACT24 ist bekannt, daß mehrere Konfigurationen entweder zugleich konfiguriert werden können und danach durch Trigger (PACT08) angesteuert werden oder nur eine Untermenge konfiguriert ist und die restlichen Konfigurationen bei Bedarf dadurch nachgeladen werden, daß die entsprechenden Trigger an eine Ladeeinheit (CT, PACT10) gesendet werden.

Der im folgenden gebrauchte Wert  $PAR(p)$  gibt zur Verdeutlichung an, welche Parallelität auf Instruktionsniveau, d. h.

wieviel ILP bei einer bestimmten Stufe (p) innerhalb des aus der Funktion transformierten Datenflußgraphen erreichbar ist (Figur 7a).

Gleichfalls bedeutsam ist Vektorparallelität (vgl. VEC{}). Vektorparallelität ist nutzbar, wenn größere Datenmengen zu verarbeiten sind. In diesem Fall sind lineare Folgen von Operationen vektorisierbar, d.h. alle Operationen können gleichzeitig Datenverarbeiten, wobei typischerweise jede separate Operation ein separates Datenwort bearbeitet.

Innerhalb von Schleifen ist dieses Vorgehen teilweise nicht möglich. Daher sind Analysen und Optimierungen notwendig. Beispielsweise kann der Graph einer Funktion durch ein Petri-netz ausgedrückt werden. Petri-Netze besitzen die Eigenschaft, daß die Ergebnisweitergabe von Knoten kontrolliert erfolgt, wodurch beispielsweise Schleifen modelliert werden können. Durch die Rückkopplung des Ergebnisses in einer Schleife wird der Datendurchsatz bestimmt. Beispiele:

- Das Ergebnis der Berechnung  $n$  wird für die Berechnung  $n+1$  benötigt: Nur eine Berechnung kann innerhalb der Schleife ausgeführt werden.
- Das Ergebnis der Berechnung  $n$  wird für die Berechnung  $n+m$  benötigt:  $m-1$  Berechnungen können innerhalb der Schleife ausgeführt werden.
- Das Ergebnis bestimmt den Abbruch der Schleife, geht aber nicht in die Berechnung der Ergebnisse ein: Eine Rückkopplung ist nicht notwendig. Ggf. laufen zwar falsche (zuviel) Werte in die Schleife, jedoch kann die Ausgabe der Ergebnisse direkt bei Erreichen der Endbedingung am Schleifenende unterbrochen werden.

Vor der Analyse von Schleifen können diese nach dem Stand der Technik optimiert werden. Beispielsweise können bestimmte In-

struktionen aus der Schleife herausgezogen werden und vor oder nach die Schleife gestellt werden.

Der im Folgenden zur Verdeutlichung gebrauchte Wert VEC kann den Grad der Vektorisierbarkeit einer Funktion veranschaulichen. Mit anderen Worten zeigt VEC an, wieviele Datenworte zugleich innerhalb einer Menge von Operationen bearbeitet werden können. VEC kann beispielsweise aus der Zahl der benötigten Rechenwerke für eine Funktion  $n_{\text{nodes}}$  und der zugleich innerhalb des Vektors berechenbaren Daten  $n_{\text{data}}$  berechnet werden, z.B. durch 
$$\text{VEC} = n_{\text{nodes}} / n_{\text{data}}$$

Ist eine Funktion beispielsweise auf 5 Rechenwerke abbildbar ( $n_{\text{nodes}} = 5$ ) und in jedem der Rechenwerke können zugleich Daten bearbeitet werden ( $n_{\text{data}} = 5$ ) ist  $\text{VEC} = 1$  (Figur 7b).

Ist eine Funktion dagegen beispielsweise auf 5 Rechenwerke abbildbar ( $n_{\text{nodes}} = 5$ ) und nur in einem Rechenwerk können jeweils Daten bearbeitet werden, z. B. aufgrund einer Rückkopplung der Ergebnisse der Pipeline auf den Eingang ( $n_{\text{data}} = 5$ ), so ist  $\text{VEC} = 1/5$  (Figur 7c).

VEC kann für eine gesamte Funktion und/oder für Teilausschnitte einer Funktion berechnet werden. Für den erfindungsgemäßen Compiler können beide Varianten vorteilhaft sein, wie generell die Bestimmung und Auswertung von VEC vorteilhaft ist.

Gemäß Figur 7a wird  $\text{PAR}(p)$  für jede Zeile eines Graphen bestimmt, wie vorteilhaft möglich. Eine Zeile eines Graphen ist dadurch definiert, daß sie innerhalb einer Takteinheit ausgeführt wird. Die Anzahl der Operationen ist von der Implementierung der jeweiligen VPU abhängig.

Entspricht  $\text{PAR}(p)$  der Anzahl der Knoten in der Zeile  $p$ , so können alle Knoten parallel ausgeführt werden.

Ist PAR(p) kleiner, werden bestimmte Knoten nur alternativ ausgeführt. Die alternativen Ausführungen jeweils eines Knotens werden in jeweils einer PAE zusammengefaßt. Eine Selektionsvorrichtung ermöglicht die Aktivierung der, dem Status der Datenverarbeitung entsprechenden, Alternative zur Laufzeit wie beispielsweise in PACT08 beschrieben.

VEC wird ebenfalls jeder Zeile eines Graphen zugeordnet. Ist für eine Zeile VEC = 1, bedeutet dies, daß die Zeile als Pipelinestufe bestehen bleibt. Ist eine Zeile kleiner 1, so werden alle nachfolgenden Zeilen, die ebenfalls kleiner 1 sind zusammengefaßt, da ein Pipelining nicht möglich ist. Entsprechend der Reihenfolge der Operationen werden diese zu einer Sequenz zusammengefaßt, die dann in eine PAE konfiguriert wird und zur Laufzeit sequentiell abgearbeitet wird. Entsprechende Verfahren sind beispielsweise aus PCT/DE 97/02949 und/oder PCT/DE 97/02998 bekannt.

Durch das beschriebene Verfahren lassen sich durch Gruppierungen von Sequenzen beliebig komplexe Parallelprozessormodelle aufbauen. Insbesondere sind Sequenzerstrukturen zur Abbildung von reentrantem Code generierbar.

Die dazu jeweils notwendigen Synchronisationen können beispielsweise durch das in PACT18 beschriebene TimeStamp-Verfahren oder bevorzugt durch das in PACT08 beschriebene Triggerverfahren durchgeführt werden.

Werden mehrere Sequenzer oder sequentielle Teile auf ein PA abgebildet, ist es aus Leistungsverbrauchsgründen bevorzugt, die Leistung der einzelnen Sequenzer aufeinander abzustimmen. Dies kann besonders bevorzugt derart geschehen, daß die Arbeitsfrequenzen der Sequenzer aneinander angepaßt werden. Aus PACT25 und PACT18 sind beispielsweise Verfahren bekannt, die eine individuelle Taktung von einzelnen PAEs oder PAE-Gruppen zulassen.



Die Frequenz eines Sequenzers kann dabei anhand der Anzahl von Zyklen bestimmt werden, die er typischerweise zur Abarbeitung der ihm zugewiesenen Funktion benötigt.

Benötigt er beispielsweise 5 Taktzyklen zur Abarbeitung seiner Funktion während das restliche System genau einen Taktzyklus benötigt, um zugewiesene Aufgaben abzuarbeiten, sollte seine Taktung 5-mal höher sein als die Taktung des restlichen Systems. Bei einer Vielzahl von Sequenzern sind jeweils unterschiedliche Taktzyklen möglich. Es kann eine Taktvervielfachung und/oder eine Taktteilung vorgesehen werden.

Funktionen werden entsprechend des vorgenannten Verfahrens partitioniert. Beim Partitionieren werden entsprechend Speicher für Daten und relevanten Status eingefügt. Weitere alternative und/oder weitergehende Verfahren sind aus PACT13 und PACT18 bekannt.

Manche VPU's bieten nach PACT01, PACT10, PACT13, PACT17, PACT22, PACT24 die Möglichkeit der differentiellen Rekonfiguration. Diese kann angewendet werden, wenn nur verhältnismäßig wenige Änderungen innerhalb der Anordnung von PAEs bei einer Rekonfiguration notwendig werden. Mit anderen Worten werden nur die Veränderungen einer Konfiguration gegenüber der aktuellen Konfiguration rekonfiguriert. Die Partitionierung kann in diesem Fall dergestalt sein, daß die auf eine Konfiguration folgenden (differentielle) Konfiguration nur die notwendigen Rekonfigurationsdaten enthält und keine vollständige Konfiguration darstellt. Der Compiler der vorliegenden Erfindung ist bevorzugt dazu ausgebildet, dies zu erkennen und zu unterstützen.

Das Scheduling der Rekonfiguration kann durch den Status erfolgen, der Funktion(en) an eine Ladeeinheit (CT) meldet, welche ihrerseits auf Basis des eingehenden Status die nächste Konfiguration oder Teilkonfiguration auswählt und konfigu-

riert. Im Detail sind derartige Verfahren aus PACT01, PACT05, PACT10, PACT13, PACT17 bekannt.

Weiterhin kann das Scheduling die Möglichkeit des Vorladens von Konfigurationen während der Laufzeit einer anderen Konfiguration unterstützen. Dabei können mehrere Konfigurationen möglicherweise auch spekulativ vorgeladen werden, d.h. ohne daß sichergestellt ist, daß die Konfigurationen überhaupt benötigt werden. Dies ist besonders dann bevorzugt, wenn die CT etwa bei längeren, konfigurationsfrei abarbeitbaren Datenströmen zumindest weitgehend unbelastet ist und insbesondere nicht oder nur wenig aufgabenbelastet ist. Durch Selektionsmechanismen wie etwa nach DE 197 04 728.9 werden dann zur Laufzeit die zu verwendenden Konfigurationen ausgewählt (siehe auch Beispiel NLS in PACT22/24).

Ebenfalls können die lokalen Sequenzer durch den Status ihrer Datenverarbeitung gesteuert werden, wie etwa aus DE 196 51 075.9-53, DE 196 54 846.2-53, DE 199 26 538.0 bekannt. Zur Durchführung ihrer Rekonfiguration kann ein weiterer abhängiger oder unabhängiger Status an die CT gemeldet werden (siehe beispielsweise PACT04, LLBACK).

Das Vorstehende wird nun mit Bezug auf weitere Figuren beschrieben. Dabei werden im folgenden folgende Zeichen zur Vereinfachung der Schreibung verwendet:  $\vee$  oder,  $\wedge$  und.

**Figur 8a** zeigt die Abbildung des Graphens nach Fig. 7a auf eine Gruppe von PAEs bei maximaler erreichbarer Parallelität. Sämtliche Operationen (Instruktion i1-i12) sind in einzelne PAEs abgebildet.

**Figur 8b** zeigt denselben Graphen, beispielsweise mit maximaler nutzbarer Vektorisierbarkeit. Jedoch sind die Mengen von Operationen  $V2=\{i1, i3\}$ ,  $V3=\{i4, i5, i6, i7, i8\}$ ,  $V4=\{i9, i10, i11\}$  nicht parallel par ( $\text{par}(\{2,3,4\})=1$ ). Damit lassen sich

Ressourcen einsparen, indem jeweils eine Menge P2, P3, P4 von Operationen einer PAE zugeordnet wird. Ein Statussignal zu jedem Datenwort in jeder Stufe wählt die auszuführende Operation in der jeweiligen PAE aus. Die PAEs sind als Pipeline (Vektor) vernetzt und jede PAE führt je Takt eine Operation über jeweils unterschiedliche Datenwort aus.

Es ergibt sich folgender Ablauf:

PAE1 berechnet Daten und gibt diese an PAE2 weiter. Zusammen mit den Daten gibt sie ein Statussignal weiter, das anzeigt, ob i1 oder i2 ausgeführt werden soll.

PAE2 berechnet die Daten von PAE1 weiter. Entsprechend des eingehenden Statussignals wird die auszuführende Operation (i1, i2) ausgewählt und berechnet. Entsprechend der Berechnung gibt PAE2 ein Statussignal an PAE3 weiter, das anzeigt, ob (i4  $\vee$  i5)  $\vee$  (i6  $\vee$  i7  $\vee$  i8) ausgeführt werden soll.

PAE3 berechnet die Daten von PAE2 weiter. Entsprechend des eingehenden Statussignals wird die auszuführende Operation (i4  $\vee$  i5)  $\vee$  (i6  $\vee$  i7  $\vee$  i8) ausgewählt und berechnet. Entsprechend der Berechnung gibt PAE3 ein Statussignal an PAE4 weiter, das anzeigt ob i9  $\vee$  i10  $\vee$  i11 ausgeführt werden soll.

PAE4 berechnet die Daten von PAE3 weiter. Entsprechend des eingehenden Statussignals wird die auszuführende Operation i9  $\vee$  i10  $\vee$  i11 ausgewählt und berechnet.

PAE5 berechnet die Daten von PAE4 weiter.

Ein mögliches entsprechendes Verfahren und Hardware, die eine besonders günstige Umsetzung des beschriebenen erlaubt, ist in DE 197 04 728.9 (Figuren 5 und 6) beschrieben; auch PACT04 und PACT10, PACT13 beschreiben allgemein nutzbare, jedoch aufwendigere Verfahren.

**Figur 8c** zeigt wiederum denselben Graphen. In diesem Beispiel ist eine Vektorisierung nicht möglich, jedoch ist PAR(p) hoch,

was bedeutet, daß innerhalb einer Zeile jeweils eine Vielzahl von Operationen gleichzeitig ausgeführt werden kann. Die parallel durchführbaren Operationen sind  $P2 = \{i1 \wedge i2\}$ ,  $P3 = \{i4 \wedge i5 \wedge i6 \wedge i7 \wedge i8\}$ ,  $P4 = \{i9 \wedge i10 \wedge i11\}$ . Die PAEs sind derart vernetzt, daß sie beliebige Daten beliebig untereinander austauschen können. Die einzelnen PAEs führen nur dann Operationen durch, wenn im entsprechenden Zyklus ein ILP besteht, ansonsten verhalten sie sich neutral (NOP), wobei ggf. Heruntertaktung und/oder eine Takt- und/oder Stromabschaltung zur Minimierung der Verlustleistung erfolgen kann.

Es ist dabei folgender Ablauf vorgesehen:

Im ersten Zyklus arbeitet nur PAE2 und gibt die Daten an PAE2 und PAE3 weiter.

Im zweiten Zyklus arbeiten PAE2 und PAE3 parallel und geben ihre Daten an PAE1, PAE2, PAE3, PAE4, PAE5 weiter.

Im dritten Zyklus arbeiten PAE1, PAE2, PAE3, PAE4, PAE5 und geben die Daten an PAE2, PAE3, PAE5 weiter.

Im vierten Zyklus arbeiten PAE2, PAE3, PAE5 und geben die Daten an PAE2 weiter.

Im fünften Zyklus arbeitet nur PAE2.

Die Funktion benötigt somit 5 Zyklen zur Berechnung. Der entsprechende Sequenzer sollte also mit dem 5-fachen Takt im Verhältnis zu seiner Umgebung arbeiten, um eine entsprechende Performance zu erzielen.

Ein mögliches entsprechendes Verfahren ist in PACT02 (Figuren 19, 20 und 21) beschrieben; auch PACT04 und PACT10, 13 beschreiben allgemein nutzbare, jedoch aufwendigere Verfahren. Weitere Verfahren und/oder Hardware sind verwendbar.

**Figur 8d** zeigt den Graphen nach Fig. 7a für den Fall, daß keinerlei nutzbare Parallelität besteht. Zur Berechnung eines Datenwortes muß jede Stufe nacheinander durchlaufen werden. In-

nerhalb der Stufen wird immer nur genau einer der Zweige verarbeitet.

Die Funktion benötigt ebenfalls 5 Zyklen zur Berechnung,  $cy1 = (i1)$ ,  $cy2 = (i2 \vee i3)$ ,  $cy3 = (i4 \vee i5 \vee i6 \vee i7 \vee i8)$ ,  $cy4 = (i9 \vee i10 \vee i11)$ ,  $cy5 = (i12)$ . Der entsprechende Sequenzer sollte also mit dem 5-fachen Takt im Verhältnis zu seiner Umgebung arbeiten, um eine entsprechende Performance zu erzielen.

Eine derartige Funktion ist beispielsweise ähnlich Fig. 8c durch einen einfachen Sequenzer nach PACT02 (Figuren 19, 20 und 21) abbildbar. Auch PACT04 und PACT10, 13 beschreiben allgemein nutzbare, jedoch aufwendigere Verfahren.

Die in Figur 8 dargestellten Abbildungen sind beliebig mischbar und gruppierbar.

In **Figur 9a** ist beispielsweise dieselbe Funktion dargestellt, bei welcher die Pfade  $(i2 \wedge (i4 \vee i5) \wedge i9)$  und  $(i3 \wedge (i6 \vee i7 \vee i8) \wedge (i9 \vee i10))$  parallel ausführbar sind.  $(i4 \vee i5)$ ,  $i6 \vee i7 \vee i8$ ,  $(i9 \vee i10)$  sind jeweils alternativ. Die Funktion ist weiterhin vektorisierbar. Damit läßt sich eine Pipeline aufbauen, in welcher für 3 PAEs (PAE4, PAE5, PAE7) jeweils anhand von Statussignalen die jeweilig auszuführende Funktion bestimmt ist.

**Figur 9b** zeigt ein ähnliches Beispiel, bei dem eine Vektorisierung nicht möglich ist. Allerdings sind die Pfade  $(i1 \wedge i2 \wedge (i4 \vee i5) \wedge i9 \wedge i12)$  und  $(i3 \wedge (i6 \vee i7 \vee i8) \wedge (i10 \vee i11))$  parallel. Damit läßt sich die optimale Performance durch den Einsatz von zwei PAEs erzielen, die bei den parallelen Pfaden auch parallel abarbeiten. Die Synchronisation der PAEs untereinander erfolgt durch Statussignale, die vorzugsweise von PAE1 generiert werden, da diese den Beginn  $(i1)$  und das Ende  $(i12)$  der Funktion berechnet.

Es soll besonders darauf hingewiesen werden, daß sich aus einer mehrfachen Anordnung von Sequenzern ein symmetrisch paralleles Prozessormodell (SMP) oder ähnliche, heute verwendete Mehrprozessormodelle ergeben können.

Weiterhin soll darauf hingewiesen werden, daß sämtliche Konfigurationsregister für das Scheduling auch im Hintergrund und/oder während der Datenverarbeitung mit neuen Konfigurationen geladen werden können.

Es ist dies etwa möglich, wenn die Hardware wie nach DE 196 51 075.9-53 bekannt aufgebaut ist. Es stehen dann unabhängige Speicherbereiche oder Register zur Verfügung, die unabhängig angesprochen werden können. Auf bestimmte Stellen wird durch eingehende Trigger gesprungen, ebenfalls kann mittels Sprungbefehlen (JMP, CALL/RET), die ggf. auch bedingt durchführbar sind gesprungen werden.

Gemäß DE 196 54 846.2-53 stehen unabhängige Schreib- und Lesezeiger zur Verfügung, wodurch grundsätzlich eine Unabhängigkeit und somit die Möglichkeit des Zugriffes im Hintergrund gegeben ist. Insbesondere ist es möglich, die Speicher zu segmentieren, wodurch eine zusätzliche Unabhängigkeit gegeben ist. Mittels Sprungbefehlen (JMP, CALL/RET), die ggf. auch bedingt durchführbar sind, kann gesprungen werden.

Nach DE 197 04 728.9 sind die einzelnen Register, die durch die Trigger gewählt werden können, grundsätzlich unabhängig und erlauben daher eine unabhängige Konfiguration, insbesondere im Hintergrund. Sprünge innerhalb der Register sind nicht möglich, die Auswahl erfolgt ausschließlich über die Triggervektoren.

Ein wesentlicher Faktor zur Bewertung der Effizienz von PAR und VEC ist die Art der Daten die durch die jeweilige Struktur verarbeitet werden. Beispielsweise ist es lohnend eine Struktur auszuwalzen, also zu pipelinieren und oder parallelisieren, die eine große Menge von Daten verarbeitet; wie es z.B. bei Videodaten oder Telekomdaten der Fall ist. Strukturen die wenige Daten verarbeiten (z.B. Tastatureingabe, Maus, etc.) lohnen sich nicht ausgewalzt zu werden, im Gegenteil sie würden nur anderen Algorithmen die Ressourcen blockieren.

Somit wird vorgeschlagen anhand unterschiedlicher Hinweise nur die Algorithmen, Strukturen oder Teile von Algorithmen zu parallelisieren und vektorisieren, die entsprechend große Datenmengen Verarbeiten.

Derartige Hinweise können beispielsweise sein:

1. Der Datentyp (Arrays, Streams sollten z.B. aufgrund der hohen Datenmenge eher ausgewalzt werden als z.B. einzelne Zeichen).
2. Die Art des Zugriffes (lineare Programmabfolgen sollten z.B. in Sequenzen abgebildet werden, während Schleifen sich z.B. aufgrund der hohen Anzahl von Durchläufen zum Auswalzen lohnen).
3. Die Art der Quelle und/oder des Ziels (Tastatur und Maus haben z.B. eine zu geringe Datenrate um effizient ausgewalzt zu werden, dagegen ist z.B. die Datenrate bei Netzwerk und/oder Video Quellen oder Zielen deutlich höher).

Für die Analyse können dabei eine beliebige Menge dieser Hinweise hinzugezogen werden.

## 7. Begriffsdefinition

**lokal relevanter Zustand**

Zustand, der nur innerhalb einer bestimmten Konfiguration relevant ist;

**global relevanter Zustand**

Zustand, der in mehreren Konfigurationen relevant ist und zwischen den Konfigurationen ausgetauscht werden muß;

**relevanter Zustand**

Zustand, der innerhalb eines Algorithmus zu dessen korrekter Ausführung dessen benötigt wird und somit durch den Algorithmus beschrieben ist und davon verwendet wird;

**irrelevanter Zustand**

Zustand, der für den eigentlichen Algorithmus ohne Bedeutung ist und auch nicht im Algorithmus beschrieben ist, der jedoch von der ausführenden Hardware implementierungsabhängig benötigt wird



## Patentansprüche

1. Verfahren zum Übersetzen von Hochsprachen auf rekonfigurierbare Architekturen, dadurch gekennzeichnet, daß ein endlicher Automat zur Berechnung derart aufgebaut wird, daß ein komplexes kombinatorisches Netz aus einzelnen Funktionen gebildet wird und dem Netz Speicher zur Speicherung der Operanden und Ergebnissen zugeordnet sind.
2. Verfahren zum Datenbe- und/oder verarbeitung mit einem multidimensionalen Feld mit rekonfigurierbaren ALUs, dadurch gekennzeichnet, daß ein Hochsprachencode vorgesehen und derart übersetzt wird, daß ein endlicher Automat zur Berechnung aufgebaut wird, wobei ein komplexes kombinatorisches Netz aus einzelnen Funktionen gebildet und dem Netz Speicher zur Speicherung der Operanden und/oder Ergebnisse zugeordnet werden.
3. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß das komplexe kombinatorische Netz so aufgebaut und/oder zerlegt wird, daß die PAE-Matrix möglichst lange ohne Rekonfiguration betrieben wird.
4. Verfahren nach dem vorhergehenden Anspruch, dadurch gekennzeichnet, daß komplexe Instruktionen bestimmt werden, um das komplexe kombinatorische Netz so aufzubauen und/oder zu zerlegen, daß die PAE-Matrix möglichst lange ohne Rekonfiguration betrieben wird.
5. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß ein endlicher Automat direkt aus imperativem Quelltext aufgebaut wird.

6. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß ein endlicher Automat aus an grobgranulare Logikkreise und/oder an vorhandene feingranulare Elemente (FPGA-Zellen in der VPU, statemachines etc.) angepaßte Operationen aufgebaut wird, insbesondere ausschließlich an solche.
7. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß ein endlicher Automat dann in Konfigurationen zerlegt wird.
8. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß generierte Konfigurationen successive auf die PAE-Matrix abgebildet werden und Arbeitsdaten und/oder Zustände, die zwischen den Konfigurationen zu übertragen sind, in Speicher abgelegt werden.
9. Verfahren nach dem vorhergehenden Anspruch, dadurch gekennzeichnet, daß der Speicher vom Compiler bestimmt beziehungsweise vorgesehen wird.
10. Verfahren nach dem vorhergehenden Anspruch, dadurch gekennzeichnet, daß während einer Konfiguration Daten aus einer VPU externen Quelle und/oder einem internen Speicher verarbeitet und an eine externe Quelle und/oder einen internen Speicher geschrieben werden.
11. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß ein Speicher für einen gesamten Datensatz vorgesehen wird, der umfangreicher als ein einzelnes Datenwort ist.
12. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß bei Verarbeitung einer ablaufenden Kon-

figuration Daten compilerbestimmt in die Speicher abgelegt werden.

13. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß ein Speicher für Operanden, ein Speicher für Ergebnisse und ein Netzwerk aus Zuweisungen und/oder Vergleichen-Anweisungen, also Bedingungen wie z.B. IF, CASE, Schleifen (WHILE, FOR, REPEAT) sowie optionalen Adressgenerator(en) zur Ansteuerung der Speicher mit dem Automaten vorgesehen werden.
14. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß Zuständen wie erforderlich Speicher zugeordnet werden, und hierbei zwischen algorithmisch relevanten und irrelevanten Zuständen unterschieden wird, insbesondere solchen relevanten Zuständen, die innerhalb des Algorithmus notwendig um dessen korrekte Funktion zu beschreiben und solchen irrelevanten Zustände, die durch die verwendete Hardware und/oder die gewählte Abbildung oder aus anderen sekundären Gründen entstehen.
15. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß Load/Store Operationen unter Vorsehen einer externen Adressierung, also des Datentransfers mit externen Baugruppen und einer internen Adressierung, also die Datentransfers zwischen PAEs, i.b. zwischen RAM-PAEs und ALU-PAEs vorgesehen werden.
16. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß bei der Datenverarbeitung eine erste Konfiguration entfernt wird und die zu sichernden Daten in entsprechenden Speichern (REG) (Speicher, Register, Zähler, etc) verbleiben.

17. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß die erste Konfiguration wieder geladen wird und auf die zuvor gesicherten, ihr zugeordnete Daten zugreift.
18. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß für den Zugriff auf zuvor gesicherte Daten eine zweite Konfiguration geladen wird, die die REG in geeigneter Weise und definierter Reihenfolge mit einem oder mehreren globalen Speicher(n) verbindet, insbesondere, um unter Verwendung von Adressgeneratoren auf den/die globalen Speicher zuzugreifen, wobei der Adressgenerator die Adressen für den/die globalen Speicher(n) bevorzugt derart generiert, dass die beschriebenen Speicherbereiche (PUSHAREA) der entfernten ersten Konfiguration eindeutig zugeordnet werden können.
19. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß automatisch Transformation zur Repräsentation der Parallelisier- bzw. Vektorisierbarkeit (Par-Vec-Transformation) durchgeführt werden und/oder VEC und PAR-Anteile als Petri-Netz ausgestaltet werden, um wie bevorzugt die Weiterverarbeitung nach kompletter Verarbeitung der jeweiligen Inhalte zu steuern.
20. Verfahren nach einem der vorhergehenden Ansprüche, dadurch gekennzeichnet, daß  
arithmetisch/logische Befehle direkt in das kombinatorische Netz abgebildet werden und/oder  
Sprünge (Jump/Call) entweder direkt in das kombinatorische Netz ausgewalzt und/oder durch Rekonfiguration realisiert werden und/oder  
Bedingungen und Kontrollflußbefehle (if, etc) entweder im kombinatorischen Netz vollständig aufgelöst und /oder bearbeitet werden und/oder an eine übergeordnete Konfigurati-

onseinheit weitergeleitet werden, die sodann entsprechend des entstandenen Status eine Rekonfiguration durchführt und/oder

Load/Store-Operationen in /separate Konfigurationen abgebildet und/oder durch Adressgeneratoren realisiert werden, die internen Speicher (REG{}) mittels Adressgeneratoren in externe Speicher schreiben und/oder diese von externen Speichern und/oder Peripherie laden und/oder

Register-Move-Operationen im kombinatorischen Netz durch Busse zwischen den internen Speichern (REG{}) realisiert werden und/oder

Push/Pop-Operationen durch separate Konfigurationen realisiert werden, die bestimmte interne Register im kombinatorischen Netz und/oder die internen Speicher (REG{}) mittels Adressgeneratoren in externe Speicher schreiben oder aus externen Speichern lesen und die bevorzugt vor oder nach den eigentlichen datenverarbeitenden Konfigurationen ausgeführt werden.

Fig. 1a

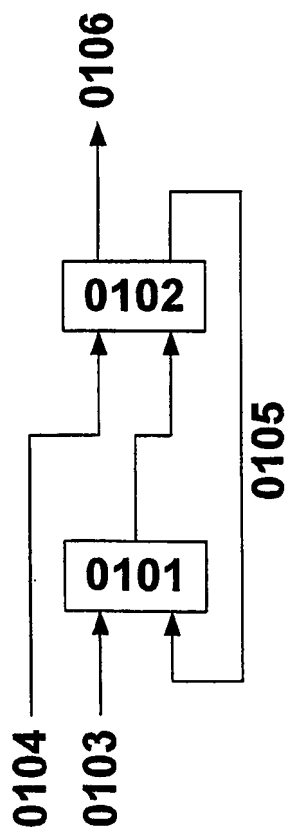
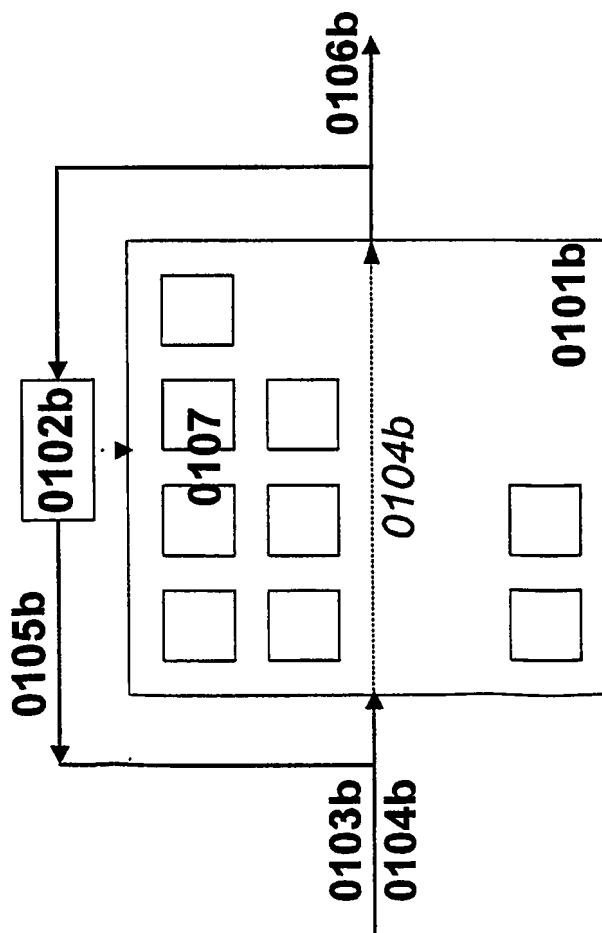


Fig. 1b



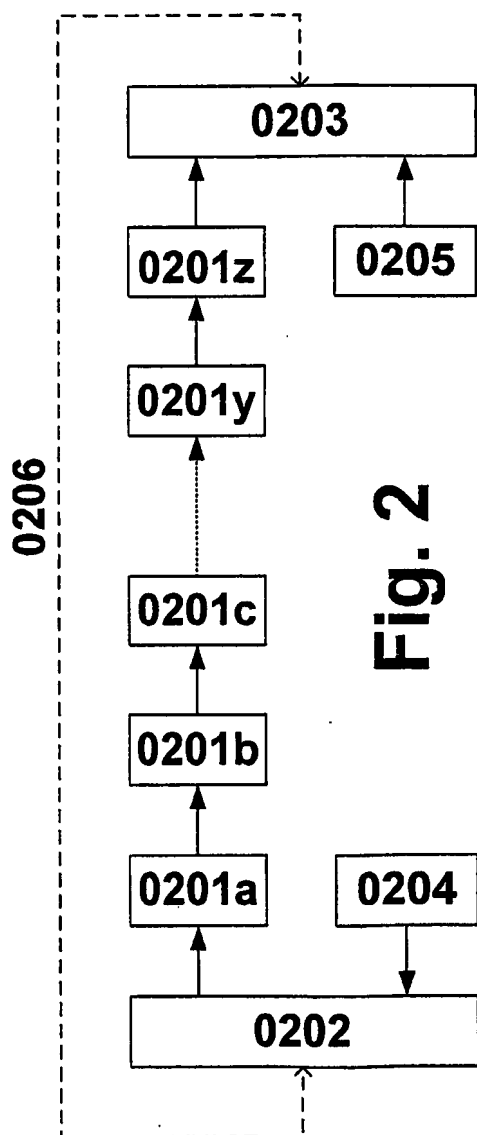


Fig. 3a

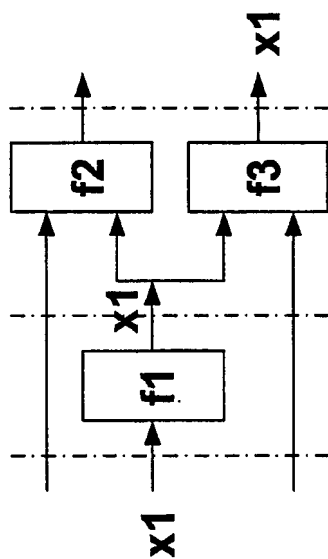


Fig. 3b

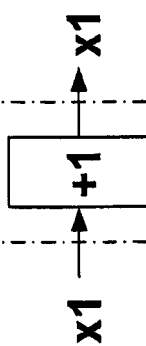
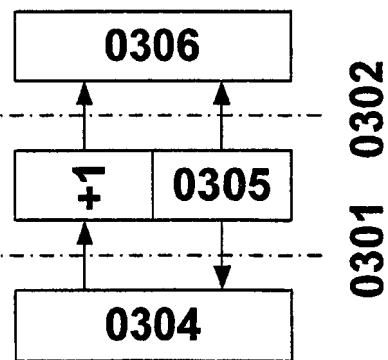
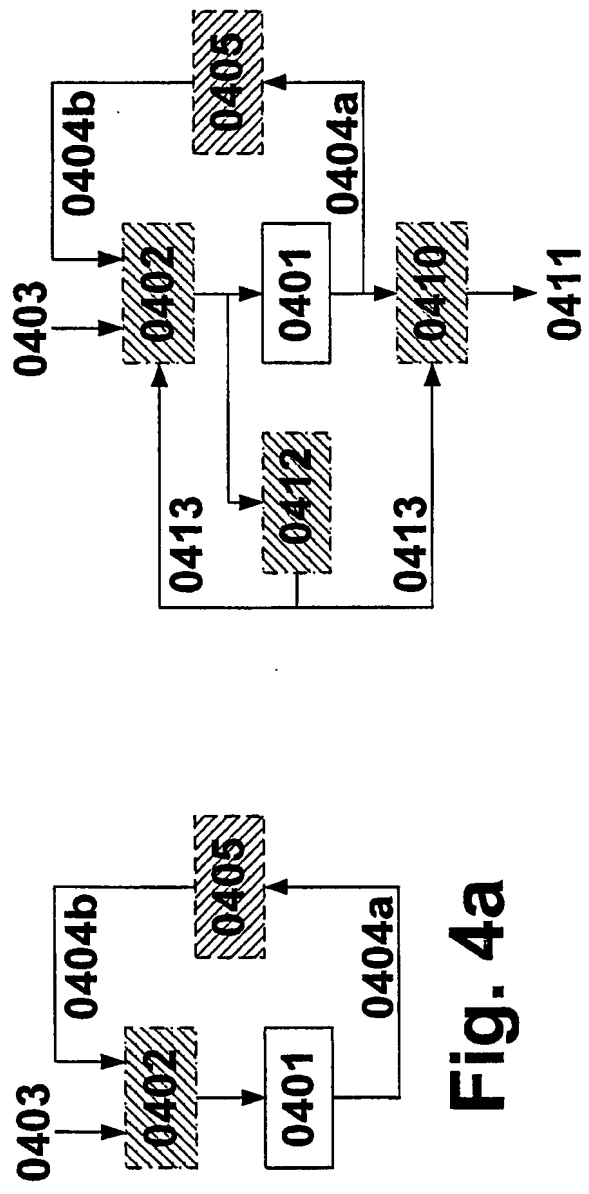


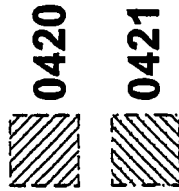
Fig. 3c







**Fig. 4a**



**Fig. 4b**

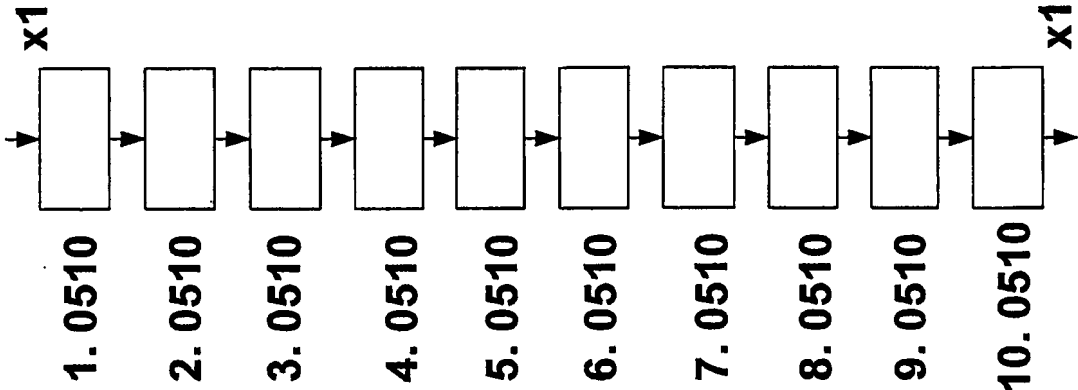


Fig. 5b

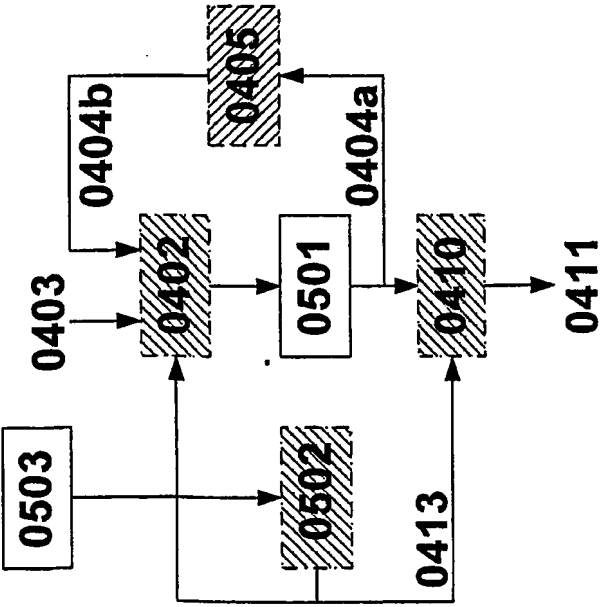
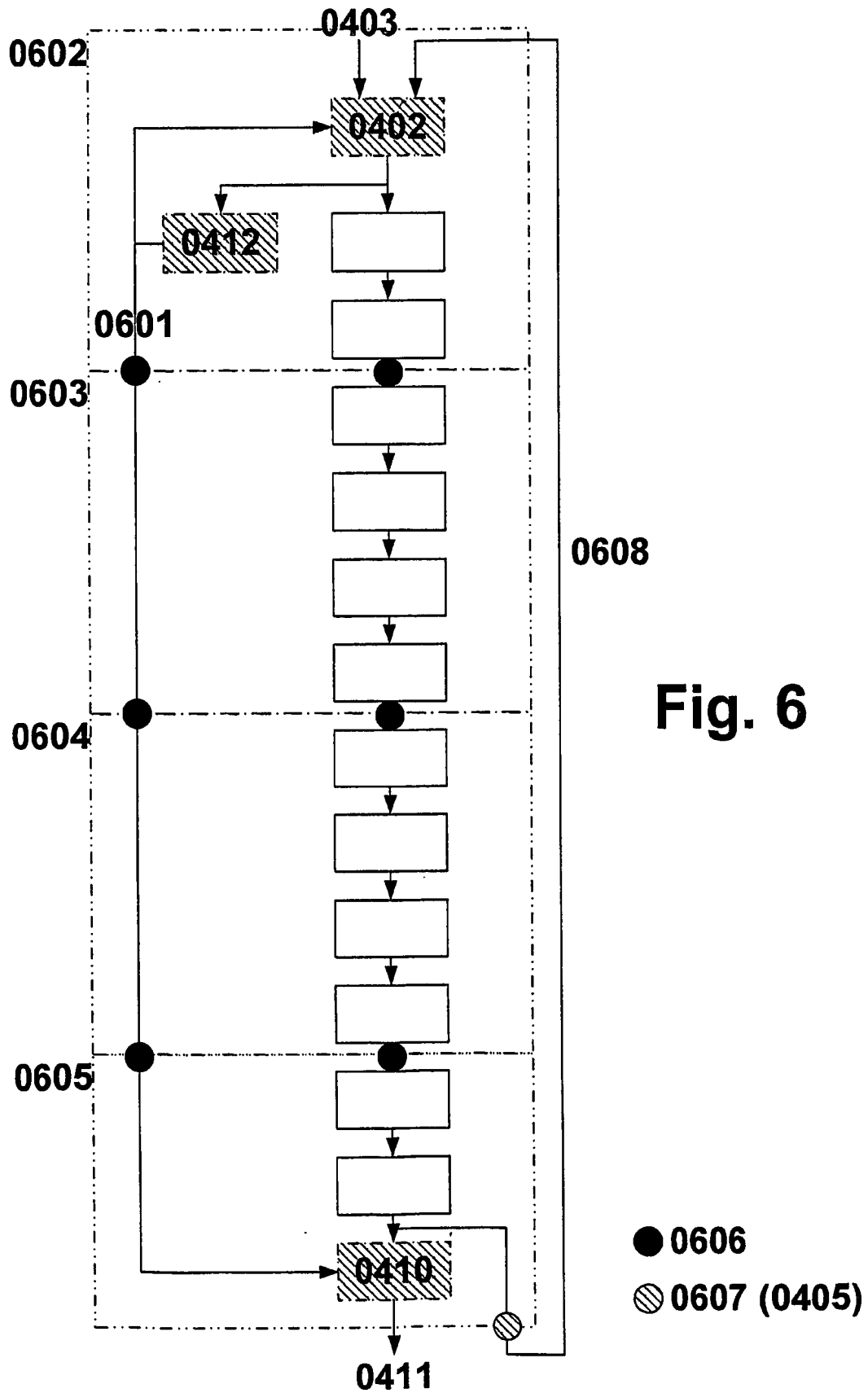


Fig. 5a



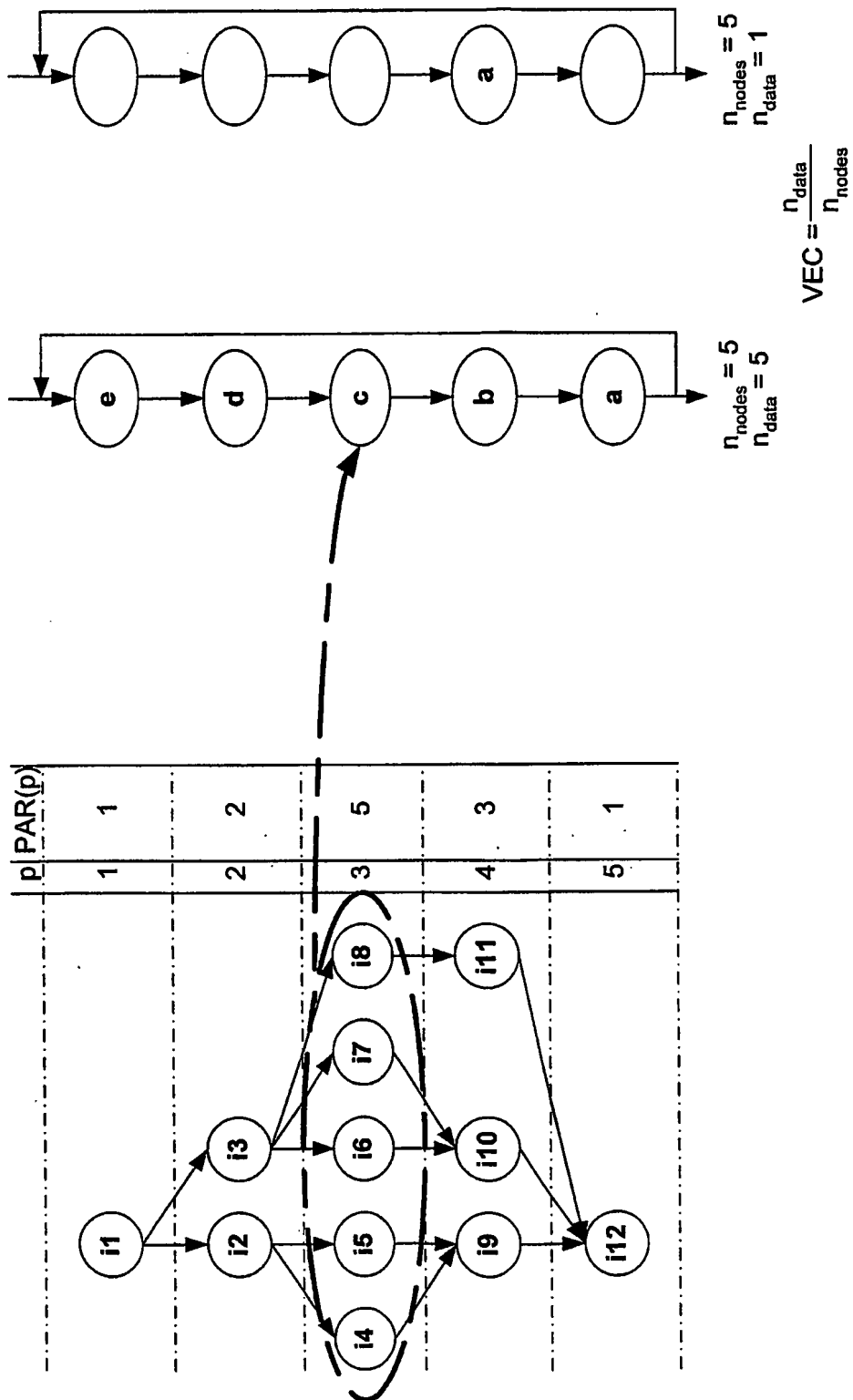


Fig. 7a Fig. 7b Fig. 7c

Fig. 7a

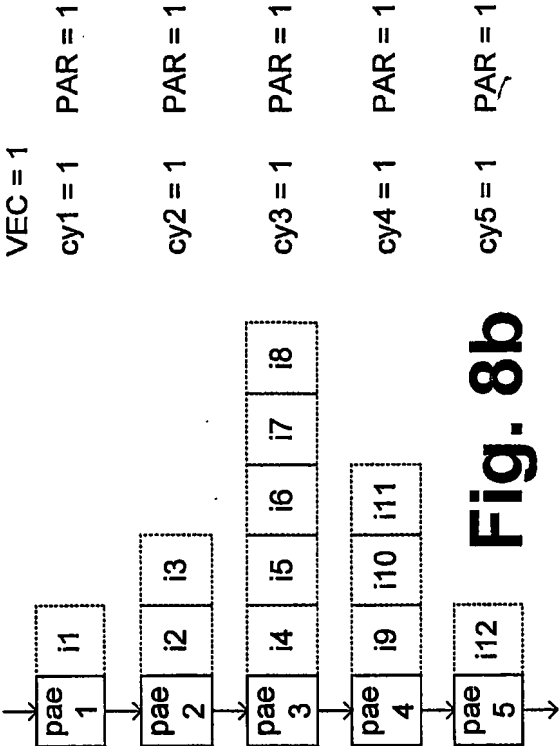


Fig. 8b

$VEC = 1/5$     $PAR(Vp) = 1$   
 $cy1 = 5$     $f_{opt} = 5 \times f$

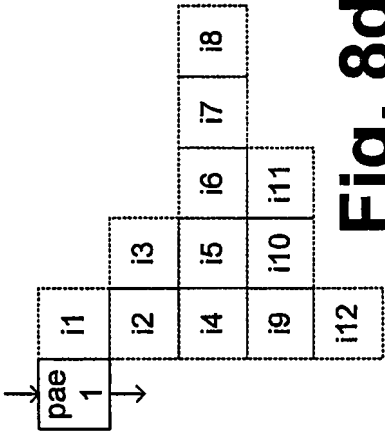


Fig. 8d

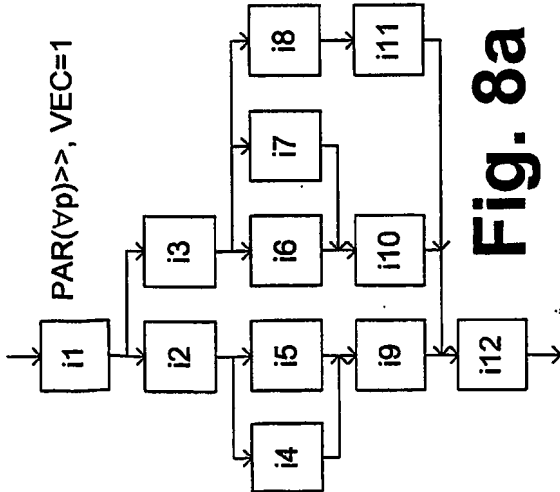


Fig. 8a

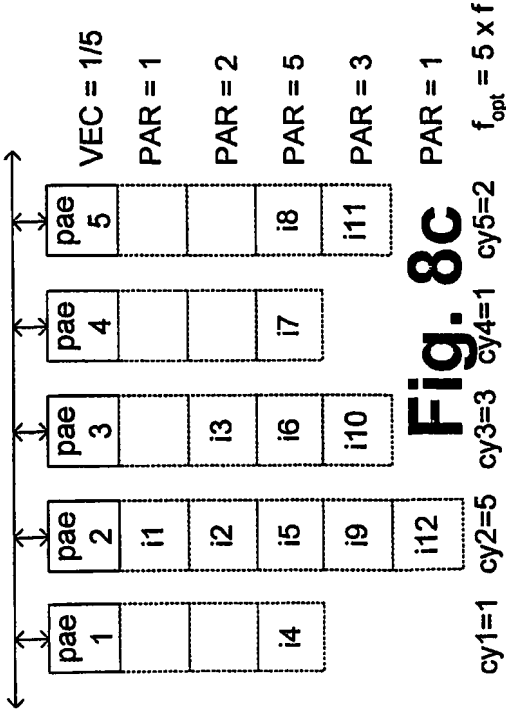


Fig. 8c

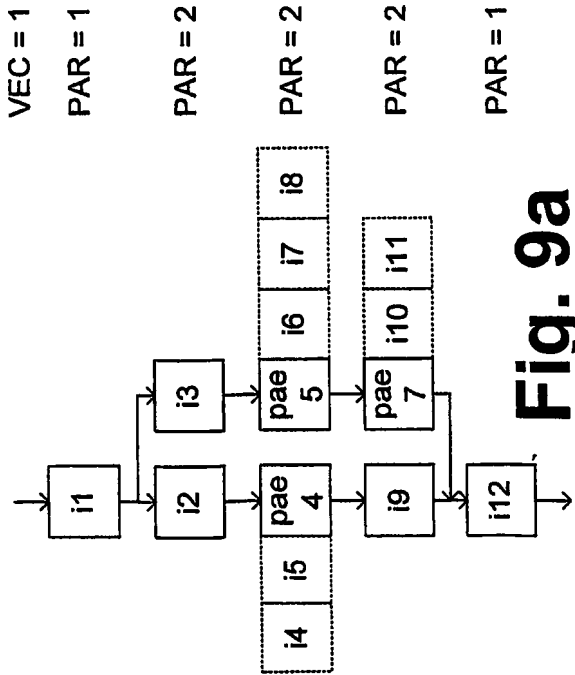


Fig. 9a

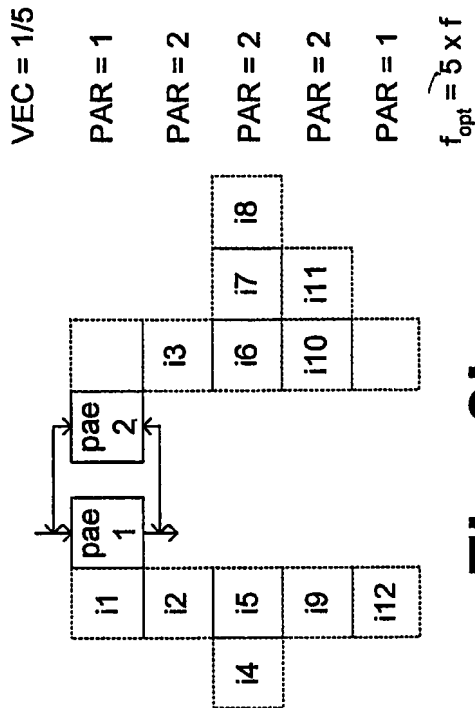


Fig. 9b